Performance Guide for IBM Informix Dynamic Server

Version 9.3
August 2001
Part No. 000-8344

© Copyright International Business Machines Corporation 2001. All rights reserved.

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i. Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick Decision Server™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Documentation Team: Virginia Panlasigui, Diane Kirsten-Martin, Cynthia Newton

Table of Contents

Introduction

In This Introduction 3
About This Manual 3
Types of Users 4
Software Dependencies 4
Assumptions About Your Locale 5
Demonstration Databases 5
New Features in Dynamic Server, Version 9.3 6
Database Server Performance Enhancements 6
Database Server Usability Enhancements 7
Sbspace Enhancements 8
Query Performance Enhancements 8
Features from Dynamic Server 9.21 9
Resolutions to Problem Tracking System (PTS) Bugs and Errata 10
Documentation Conventions 11
Typographical Conventions 11
Icon Conventions 12
Sample-Code Conventions 13
Screen-Illustration Conventions 14
Additional Documentation 14
Related Reading 17
Compliance with Industry Standards 17
Informix Welcomes Your Comments 17

Chapter 1 Performance Basics

In This Chapter 1-3
A Basic Approach to Performance Measurement and Tuning . . . 1-4
Quick Start for Small-Database User 1-5
Performance Goals 1-6
Measurements of Performance 1-7
Throughput 1-7
Response Time 1-10
Cost per Transaction 1-14
Resource Utilization and Performance 1-14
Resource Utilization 1-15
CPU Utilization 1-17
Memory Utilization 1-18
Disk Utilization 1-20
Factors That Affect Resource Utilization 1-22
Maintenance of Good Performance 1-24
Topics Beyond the Scope of This Manual 1-26

Chapter 2 Performance Monitoring

In This Chapter 2-3
Evaluating the Current Configuration 2-3
Creating a Performance History 2-4
The Importance of a Performance History 2-4
Tools That Create a Performance History 2-5
Monitoring Database Server Resources 2-11
Monitoring Resources That Impact CPU Utilization 2-11
Monitoring Memory Utilization 2-13
Monitoring Disk I/O Utilization 2-14
Monitoring Transactions 2-18
The onlog Utility 2-18
Using the onstat Utility to Monitor Transactions 2-19
Using ISA to Monitor Transactions. 2-19
Monitoring Sessions and Queries 2-20
Monitoring Memory Usage for Each Session 2-20
Using SET EXPLAIN 2-20

Chapter 3 Effect of Configuration on CPU Utilization

In This Chapter 3-3
UNIX Configuration Parameters That Affect CPU Utilization . . 3-3
UNIX Semaphore Parameters 3-4
UNIX File-Descriptor Parameters 3-6
UNIX Memory Configuration Parameters. 3-6
Windows Configuration Parameters That Affect CPU Utilization 3-7
Configuration Parameters and Environment Variables That Affect CPU Utilization 3-8
VPCLASS and Other CPU-Related Parameters 3-9
OPTCOMPIND 3-16
MAX_PDQPRIORITY 3-16
DS_MAX_QUERIES 3-17
DS_MAX_SCANS 3-18
NETTYPE 3-19
Network Buffer Pools 3-22
NETTYPE Configuration Parameter 3-24
IFX_NETBUF_PVTPOOL_SIZE Environment Variable . . . 3-24
IFX_NETBUF_SIZE Environment Variable 3-26
Virtual Processors and CPU Utilization 3-27
Adding Virtual Processors 3-27
Monitoring Virtual Processors. 3-27
Connections and CPU Utilization 3-32
Multiplexed Connections 3-32
MaxConnect for Multiple Connections 3-34

Chapter 4 Effect of Configuration on Memory Utilization

In This Chapter 4-3
Allocating Shared Memory 4-3
Resident Portion 4-4
Virtual Portion 4-6
Message Portion 4-9
Configuring UNIX Shared Memory 4-10
Freeing Shared Memory with onmode -F 4-11
Configuration Parameters That Affect Memory Utilization . . . 4-12
Setting the Size of the Buffer Pool, Logical-Log Buffer, and Physical-Log Buffer 4-13
LOCKS. 4-21
RESIDENT 4-23

SHMADD 4-24
SHMTOTAL 4-24
SHMVIRTSIZE 4-25
STACKSIZE 4-26
Parameters That Affect Memory Caches 4-27
UDR Cache 4-28
Data-Dictionary Cache 4-29
Data-Distribution Cache 4-31
SQL Statement Cache 4-37
SQL Statement Cache Configuration 4-39
Monitoring and Tuning the SQL Statement Cache 4-42
Session Memory 4-55
Data-Replication Buffers and Memory Utilization 4-56
Memory-Resident Tables and the Buffer Pool 4-56
Memory Latches 4-58
Monitoring Latches with Command-Line Utilities 4-58
Monitoring Latches with ISA 4-59
Monitoring Latches with SMI Tables 4-59

Chapter 5 Effect of Configuration on I/O Activity

In This Chapter 5-5
Chunk and Dbspace Configuration 5-5
Associate Disk Partitions with Chunks 5-6
Associate Dbspaces with Chunks 5-6
Place System Catalog Tables with Database Tables 5-7
Placement of Critical Data 5-7
Consider Separate Disks for Critical Data Components . . . 5-8
Consider Mirroring for Critical Data Components 5-8
Configuration Parameters That Affect Critical Data 5-11
Configuring Dbspaces for Temporary Tables and Sort Files 5-12
Creating Temporary Dbspaces 5-14
DBSPACETEMP Configuration Parameter 5-15
DBSPACETEMP Environment Variable 5-15
Estimating Temporary Space 5-16
PSORT_NPROCS Environment Variable 5-17
Configuring Sbspaces for Temporary Smart Large Objects 5-18
Creating Temporary Sbspaces 5-19
SBSPACETEMP Configuration Parameter 5-20
Placement of Simple Large Objects 5-20
Advantage of Blobspaces over Dbspaces. 5-21

Blobpage Size Considerations 5-22
Parameters That Affect I/O for Smart Large Objects 5-27
Disk Layout for Sbspaces 5-28
Configuration Parameters That Affect Sbspace I/O 5-28
onspaces Options That Affect Sbspace I/O 5-29
How the Optical Subsystem Affects Performance 5-34
Environment Variables and Configuration Parameters for the Optical Subsystem 5-35
STAGEBLOB. 5-36
OPCACHEMAX 5-36
INFORMIXOPCACHE 5-37
Table I/O 5-37
Sequential Scans 5-37
Light Scans 5-38
Unavailable Data 5-39
Configuration Parameters That Affect Table I/O 5-40
RA_PAGES and RA_THRESHOLD 5-40
DATASKIP 5-41
Background I/O Activities 5-41
Configuration Parameters That Affect Checkpoints 5-42
Configuration Parameters That Affect Logging 5-48
Configuration Parameters That Affect Page Cleaning. . . . 5-55
Configuration Parameters That Affect Backup and Restore . . 5-57
Configuration Parameters That Affect Rollback and Recovery. 5-58
Configuration Parameters That Affect Data Replication and Auditing 5-59

Chapter 6 Table Performance Considerations

In This Chapter 6-5
Placing Tables on Disk 6-5
Isolating High-Use Tables 6-7
Placing High-Use Tables on Middle Partitions of Disks . . . 6-8
Using Multiple Disks 6-9
Backup-and-Restore Considerations 6-11
Improving Performance for Nonfragmented Tables and Table Fragments 6-11
Estimating Table Size 6-12
Estimating Data Pages 6-12
Estimating Pages That Simple Large Objects Occupy 6-16

Managing Sbspaces 6-20
Estimating Pages That Smart Large Objects Occupy 6-20
Improving Metadata I/O for Smart Large Objects 6-23
Monitoring Sbspaces 6-25
Changing Storage Characteristics of Smart Large Objects . . . 6-30
Managing Extents 6-34
Choosing Table Extent Sizes 6-35
Monitoring Active Tblspaces. 6-38
Managing Extents 6-39
Changing Tables 6-46
Loading and Unloading Tables 6-46
Dropping Indexes for Table-Update Efficiency. 6-50
Attaching or Detaching Fragments. 6-51
Altering a Table Definition 6-51
Denormalizing the Data Model to Improve Performance 6-61
Shortening Rows 6-62
Expelling Long Strings 6-62
Splitting Wide Tables 6-64
Redundant Data 6-65

Chapter 7 Index Performance Considerations

In This Chapter 7-3
Estimating Index Pages 7-3
Index Extent Sizes 7-4
Estimating Conventional Index Pages. 7-5
Estimating Index Pages for Spatial and User-Defined Data . . 7-10
Managing Indexes 7-11
Space Costs of Indexes 7-11
Time Costs of Indexes 7-12
Choosing Columns for Indexes 7-13
Dropping Indexes 7-17
Improving Performance for Index Builds 7-18
Estimating Sort Memory 7-19
Estimating Temporary Space for Index Builds 7-20
Improving Performance for Index Checks 7-21
Indexes on User-Defined Data Types 7-22
Defining Indexes for User-Defined Data Types 7-23
Using an Index That a DataBlade Module Provides 7-31
Choosing Operator Classes for Indexes 7-31

Chapter 8 Locking

In This Chapter 8-3
Lock Granularity 8-3
Row and Key Locks 8-4
Page Locks 8-5
Table Locks 8-6
Database Locks 8-7
Configuring Lock Mode 8-7
Lock Waits 8-9
Locks with the SELECT Statement 8-9
Isolation Level 8-9
Locking Nonlogging Tables 8-12
Update Cursors 8-13
Locks Placed with INSERT, UPDATE, and DELETE 8-15
Monitoring and Administering Locks 8-15
Monitoring Locks 8-16
Configuring and Monitoring the Number of Locks 8-18
Monitoring Lock Waits and Lock Errors 8-19
Monitoring Deadlocks 8-21
Monitoring Isolation That Sessions Use 8-22
Locks for Smart Large Objects 8-22
Types of Locks on Smart Large Objects 8-23
Byte-Range Locking 8-23
Lock Promotion 8-27
Dirty Read and Smart Large Objects 8-28

Chapter 9 Fragmentation Guidelines

In This Chapter 9-3
Planning a Fragmentation Strategy 9-3
Setting Fragmentation Goals 9-4
Examining Your Data and Queries 9-8
Considering Physical Fragmentation Factors 9-9
Designing a Distribution Scheme 9-11
Choosing a Distribution Scheme 9-12
Designing an Expression-Based Distribution Scheme . . . 9-14
Suggestions for Improving Fragmentation 9-15
Fragmenting Indexes 9-17
Attached Indexes 9-17
Detached Indexes 9-18
Restrictions on Indexes for Fragmented Tables 9-20

Fragmenting Temporary Tables 9-20
Using Distribution Schemes to Eliminate Fragments 9-21
Fragmentation Expressions for Fragment Elimination 9-22
Query Expressions for Fragment Elimination 9-22
Effectiveness of Fragment Elimination 9-24
Improving the Performance of Attaching and Detaching Fragments 9-29
Improving ALTER FRAGMENT ATTACH Performance . . . 9-29
Improving ALTER FRAGMENT DETACH Performance . . . 9-37
Monitoring Fragment Use 9-39
Using the onstat Utility. 9-39
Using SET EXPLAIN 9-40

Chapter 10 Queries and the Query Optimizer

In This Chapter 10-3
The Query Plan 10-3
Access Plan. 10-4
Join Plan. 10-4
Example of Query-Plan Execution 10-8
Query Plan Evaluation 10-12
Query Plan Report 10-12
Sample Query Plan Reports 10-15
Factors That Affect the Query Plan 10-21
Statistics Held for the Table and Index 10-21
Filters in the Query 10-23
Indexes for Evaluating a Filter 10-24
Effect of PDQ on the Query Plan 10-25
Effect of OPTCOMPIND on the Query Plan 10-26
Effect of Available Memory on the Query Plan 10-27
Time Costs of a Query 10-27
Memory-Activity Costs 10-28
Sort-Time Costs 10-28
Row-Reading Costs 10-30
Sequential Access Costs 10-31
Nonsequential Access Costs 10-31
Index Lookup Costs. 10-32
In-Place ALTER TABLE Costs 10-33
View Costs 10-33
Small-Table Costs 10-34

Data-Mismatch Costs 10-35
GLS Functionality Costs 10-36
Network-Access Costs 10-36
SQL Within SPL Routines 10-38
SQL Optimization 10-38
Execution of an SPL Routine 10-41
UDR Cache 10-42
Trigger Execution 10-44
Performance Implications for Triggers 10-45

Chapter 11 Optimizer Directives

In This Chapter 11-3
Reasons to Use Optimizer Directives 11-3
Guidelines for Using Directives 11-5
Preparation for Using Directives 11-6
Types of Directives 11-7
Access-Plan Directives 11-8
Join-Order Directives 11-8
Join-Plan Directives 11-10
Optimization-Goal Directives 11-11
Example with Directives 11-12
EXPLAIN Directives 11-16
Configuration Parameters and Environment Variables for Directives 11-18
Directives and SPL Routines 11-18

Chapter 12 Parallel Database Query

In This Chapter 12-3
Structure of a Parallel Database Query 12-3
Database Server Operations That Use PDQ 12-5
Parallel Delete 12-5
Parallel Inserts 12-5
Parallel Index Builds 12-7
Parallel User-Defined Routines 12-8
SQL Operations That Do Not Use PDQ 12-8
Update Statistics 12-9
SPL Routines and Triggers 12-9
Correlated and Uncorrelated Subqueries 12-9
OUTER Index Joins 12-10
Remote Tables 12-10

Memory Grant Manager 12-10
Allocating Resources for Parallel Database Queries 12-12
Limiting the Priority of DSS Queries 12-13
Adjusting the Amount of Memory 12-17
Limiting the Number of Concurrent Scans 12-18
Limiting the Maximum Number of Queries 12-19
Managing Applications 12-19
Using SET EXPLAIN 12-19
Using OPTCOMPIND 12-20
Using SET PDQPRIORITY 12-20
User Control of Resources. 12-21
DBA Control of Resources 12-21
Monitoring PDQ Resources 12-23
Using the onstat Utility. 12-23
Using SET EXPLAIN 12-29

Chapter 13 Improving Individual Query Performance

In This Chapter 13-5
Using a Dedicated Test System 13-5
Displaying the Query Plan 13-6
Improving Filter Selectivity 13-7
Filters with User-Defined Routines 13-7
Avoiding Certain Filters 13-8
Using Join Filters and Post-Join Filters 13-9
Updating Statistics 13-13
Updating Number of Rows 13-14
Dropping Data Distributions 13-15
Creating Data Distributions 13-15
Updating Statistics for Join Columns 13-18
Updating Statistics for Columns with User-Defined Data Types 13-19
Displaying Distributions 13-20
Improving Performance of UPDATE STATISTICS 13-22
Improving Performance with Indexes 13-23
Replacing Autoindexes with Permanent Indexes 13-23
Using Composite Indexes 13-23
Using Indexes for Data Warehouse Applications 13-24
Dropping and Rebuilding Indexes After Updates 13-26

Improving Performance of Distributed Queries 13-26
Buffering Data Transfers for a Distributed Query 13-26
Displaying a Query Plan for a Distributed Query 13-27
Improving Sequential Scans 13-28
Reducing the Impact of Join and Sort Operations 13-29
Avoiding or Simplifying Sort Operations 13-29
Using Parallel Sorts 13-30
Using Temporary Tables to Reduce Sorting Scope 13-31
Optimizing User-Response Time for Queries 13-32
Optimization Level 13-32
Optimization Goal. 13-33
Optimizing Queries for User-Defined Data Types 13-37
Parallel UDRs 13-38
Selectivity and Cost Functions. 13-39
User-Defined Statistics for UDTs 13-40
Negator Functions 13-41
SQL Statement Cache 13-41
When to Use the SQL Statement Cache. 13-42
Using the SQL Statement Cache 13-43
Monitoring Memory Usage for Each Session. 13-46
Monitoring Usage of the SQL Statement Cache 13-50
Monitoring Sessions and Threads 13-52
Using Command-Line Utilities 13-52
Using ON-Monitor to Monitor Sessions 13-56
Using ISA to Monitor Sessions 13-57
Using SMI Tables 13-57
Monitoring Transactions 13-58
Displaying Transactions with onstat -x. 13-59
Displaying Locks with onstat -k 13-61
Displaying User Sessions with onstat -u 13-62
Displaying Sessions Executing SQL Statements. 13-63

Chapter 14 The onperf Utility on UNIX

In This Chapter 14-3
Overview of the onperf Utility 14-3
Basic onperf Functions 14-3
The onperf Tools 14-6
Requirements for Running onperf 14-7
Starting and Exiting onperf 14-8
The onperf User Interface 14-9
Graph Tool 14-9
Query-Tree Tool 14-19
Status Tool 14-20
Activity Tools 14-21
Ways to Use onperf 14-21
Routine Monitoring 14-21
Diagnosing Sudden Performance Loss 14-22
Diagnosing Performance Degradation 14-22
The onperf Metrics 14-22
Database Server Metrics 14-23
Disk-Chunk Metrics. 14-25
Disk-Spindle Metrics 14-26
Physical-Processor Metrics 14-26
Virtual-Processor Metrics 14-27
Session Metrics 14-27
Tblspace Metrics 14-29
Fragment Metrics. 14-30

Appendix A Case Studies and Examples

Index

Introduction

About This Manual 3
Types of Users 4
Software Dependencies 4
Assumptions About Your Locale 4
Demonstration Databases 5

New Features in Dynamic Server, Version 9.3 6
Database Server Performance Enhancements 6
Database Server Usability Enhancements 7
Sbspace Enhancements 8
Query Performance Enhancements 8
Features from Dynamic Server 9.21 9
Resolutions to Problem Tracking System (PTS) Bugs and Errata . . 10

Documentation Conventions 11
Typographical Conventions 11
Icon Conventions 12
Comment Icons 12
Feature, Product, and Platform Icons 12
Sample-Code Conventions 13
Screen-Illustration Conventions 14

Additional Documentation 14

Related Reading 16

Compliance with Industry Standards 17

Informix Welcomes Your Comments 17

In This Introduction

This introduction provides an overview of the information in this manual and describes the conventions it uses.

About This Manual

This manual provides information about how to configure and operate Informix Dynamic Server to improve overall system throughput and to improve the performance of SQL queries. Information in this manual can help you perform the following tasks:

- Monitor system resources that are critical to performance
- Identify database activities that affect these critical resources
- Identify and monitor queries that are critical to performance
- Use the database server utilities (especially **onperf,** ISA and **onstat**) for performance monitoring and tuning
- Eliminate performance bottlenecks by:
  - Balancing the load on system resources
  - Adjusting the configuration parameters or environment variables of your database server
  - Adjusting the arrangement of your data
  - Allocating resources for decision-support queries
  - Creating indexes to speed up retrieval of your data

In addition, this manual contains the full description of the **onperf** utility.

Types of Users

This manual is written for the following users:

- Database administrators
- Database server administrators
- Database-application programmers
- Performance engineers

This manual assumes that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming
- Some experience with database server administration, operating-system administration, or network administration

If you have limited experience with relational databases, SQL, or your operating system, refer to the *Getting Started* manual for your database server for a list of supplementary titles.

Software Dependencies

This manual assumes that you are using Informix Dynamic Server, Version 9.3.

Assumptions About Your Locale

Informix products can support many languages, cultures, and code sets. All the information related to character set, collation, and representation of numeric data, currency, date, and time is brought together in a single environment, called a Global Language Support (GLS) locale.

This manual assumes that you use the U.S. 8859-1 English locale as the default locale. The default is **en_us.8859-1** (ISO 8859-1) on UNIX platforms or **en_us.1252** (Microsoft 1252) for Windows NT environments. This locale supports U.S. English format conventions for date, time, and currency, and also supports the ISO 8859-1 or Microsoft 1252 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *Informix Guide to GLS Functionality*.

Demonstration Databases

The DB-Access utility, which is provided with your Informix database server products, includes one or more of the following demonstration databases:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in Informix manuals are based on the **stores_demo** database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database contains examples of extended data types, type and table inheritance, and user-defined routines.

For information about how to create and populate the demonstration databases, see the *DB-Access User's Manual*. For descriptions of the databases and their contents, see the *Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory on UNIX platforms and in the **%INFORMIXDIR%\bin** directory in Windows environments.

New Features in Dynamic Server, Version 9.3

The following tables provides information about the new features for Informix Dynamic Server, Version 9.3, that this manual covers. To go to the desired page, click a blue hyperlink. For a description of all new features, see the *Getting Started* manual.

Database Server Performance Enhancements

Version 9.3 includes many new features that help you monitor and improve the overall performance of your database server.

New Features	Reference
Configurable default lock mode	"Configuring Lock Mode" on page 8-7
Dynamically allocated log files	"Configuration Parameters That Affect Critical Data" on page 5-11 "Configuration Parameters That Affect Checkpoints" on page 5-42 "Configuration Parameters That Affect Logging" on page 5-48 "LOGSIZE" on page 5-49 "DYNAMIC_LOGS" on page 5-52 "LTXHWM and LTXEHWM" on page 5-54
Enterprise Replication enhancements: add or drop CRCOLS	"When the Database Server Uses the In-Place Alter Algorithm" on page 6-53

Database Server Usability Enhancements

Version 9.3 includes new features that make the database server easier to install, use, and manage.

New Features	Reference
Microsoft Transaction Server / XA support	"Displaying Transactions with onstat -x" on page 13-59 "Displaying User Sessions with onstat -u" on page 13-62
Informix Server Administrator (ISA) enhancements	"Informix Server Administrator" on page 2-8 "Using ISA to Monitor I/O Utilization" on page 2-15 "Using ISA to Monitor Transactions" on page 2-19 "Updating Statistics" on page 13-13 "Using ISA to Monitor Sessions" on page 13-57
Use the VPCLASS configuration parameter instead of the AFF_NPROCS, AFF_SPROC, NOAGE, NUMAIOVPS, and NUMCPUVPS configuration parameters.	"Configuration Parameters and Environment Variables That Affect CPU Utilization" on page 3-8 "VPCLASS and Other CPU-Related Parameters" on page 3-9

Sbspace Enhancements

Version 9.3 includes the following improvements for smart large objects stored in sbspaces.

New Features	Reference
Improved space allocation of user data and metadata in sbspaces	"Estimating the Size of the Sbspace and Metadata Area" on page 6-20 "Sizing the Metadata Area Manually for a New Chunk" on page 6-21 "Monitoring Sbspaces" on page 6-25
Temporary smart large objects and temporary sbspaces	"Configuring Sbspaces for Temporary Smart Large Objects" on page 5-18

Query Performance Enhancements

Version 9.3 includes several new features that help you monitor and improve the performance of individual queries.

New Features	Reference
Display memory used by SQL statements: **onstat -g stm** option	"Virtual Portion" on page 4-6 "Session Memory" on page 4-54 "onstat -g stm session-id" on page 13-49 "onstat -g mem and onstat -g stm" on page 13-55
Display Query Plan without executing the query	"Query Plan Report" on page 10-12 "EXPLAIN Directives" on page 11-16 "Displaying the Query Plan" on page 13-6

Features from Dynamic Server 9.21

These features were introduced in Informix Dynamic Server, Version 9.21.

Features	Reference
ANSI outer join	"Using Join Filters and Post-Join Filters" on page 13-9
RENAME INDEX	"Automatic Reoptimization" on page 10-40
Nonlogging (RAW) tables	"Loading and Unloading Tables" on page 6-46 "Locking Nonlogging Tables" on page 8-12
SQL statement cache enhancements	"Virtual Portion" on page 4-6 "SQL Statement Cache" on page 4-37 "SQL Statement Cache" on page 13-41

Resolutions to Problem Tracking System (PTS) Bugs and Errata

These resolutions are included in this version of this *Performance Guide*.

Features	Reference
Distributed queries clarifications for built-in data types	"Network-Access Costs" on page 10-36 "Improving Performance of Distributed Queries" on page 13-26
Indexes bugs and errata	"Index Extent Sizes" on page 7-4 "Estimating Extent Size of Detached Index" on page 7-4 "Using a Functional Index" on page 7-28
Locking errata	"Configuring and Monitoring the Number of Locks" on page 8-18
UPDATE STATISTICS PTS bug and errata	"Displaying the Execution Plan" on page 10-39 "Reoptimizing SPL Routines" on page 10-40 "Updating Statistics" on page 13-13 "Displaying Distributions" on page 13-20
Log sizes	"Estimating Logical-Log Size When Logging Dbspaces" on page 5-50

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set.

Typographical Conventions

This manual uses the following conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics *italics* `italics`	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace` *monospace*	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of one or more product- or platform-specific paragraphs.
→	This symbol indicates a menu item. For example, "Choose **Tools→Options**" means choose the **Options** item from the **Tools** menu.

Tip: *When you are instructed to "enter" characters or to "execute" a command, immediately press* RETURN *after the entry. When you are instructed to "type" the text or to "press" other keys, no* RETURN *is required.*

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.

Comment Icons

Comment icons identify three types of information, as the following table describes. This information always appears in italics.

Icon	Label	Description
	Warning:	Identifies paragraphs that contain vital instructions, cautions, or critical information
	Important:	Identifies paragraphs that contain significant information about the feature or operation that is being described
	Tip:	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described

Feature, Product, and Platform Icons

Feature, product, and platform icons identify paragraphs that contain feature-specific, product-specific, or platform-specific information.

Icon	Description
DB	Identifies information that is specific to DB-Access
DB API	Identifies information that is specific to the DataBlade API
E/C	Identifies information that is specific to Informix ESQL / C

(1 of 2)

Icon	Description
GLS	Identifies information that relates to the Informix Global Language Support (GLS) feature
UNIX	Identifies information that is specific to UNIX platforms
WIN NT	Identifies information that is specific to the Windows NT environment

(2 of 2)

These icons can apply to an entire section or to one or more paragraphs within a section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of feature-, product-, or platform-specific information that appears within one or more paragraphs within a section.

Sample-Code Conventions

Examples of SQL code occur throughout this manual. Except where noted, the code is not specific to any single Informix application development tool. If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using DB-Access, you must delimit multiple statements with semicolons. If you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement.

Tip: *Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.*

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the manual for your product.

Screen-Illustration Conventions

The illustrations in this manual represent a generic rendition of various windowing environments. The details of dialog boxes, controls, and windows were deleted or redesigned to provide this generic look. Therefore, the illustrations in this manual depict your product interface a little differently than the way it appears on your screen.

Additional Documentation

Informix Dynamic Server documentation is provided in a variety of formats:

- **Online manuals.** The Informix OnLine Documentation Web site at http://www.informix.com/answers contains manuals that Informix provides for your use. This Web site enables you to print chapters or entire books.
- **Online help.** Informix provides online help with each graphical user interface (GUI) that displays information about those interfaces and the functions that they perform. Use the help facilities that each GUI provides to display the online help.

  This facility can provide context-sensitive help, an error message reference, language syntax, and more. To order a printed manual, call 1-800-331-1763 or send email to moreinfo@informix.com. Provide the following information when you place your order:

  - The documentation that you need
  - The quantity that you need
  - Your name, address, and telephone number

- **Documentation notes.** Documentation notes, which contain additions and corrections to the manuals, are also located at the OnLine Documentation site at http://www.informix.com/answers. Examine these files before you begin using your database server.

  Release notes. Release notes contain vital information about application and performance issues. These files are located at http://www.informix.com/informix/services/techinfo. This site is a password controlled site. Examine these files before you begin using your database server.

  Documentation notes, release notes, and machine notes are also located in the directory where the product is installed. The following table describes these files.

  UNIX

  On UNIX platforms, the following online files appear in the **$INFORMIXDIR/release/en_us/0333** directory.

Online File	Purpose
performance_docnotes_9.30.html	The documentation notes file for your version of this manual describes topics that are not covered in the manual or that were modified since publication.
release_notes_9.30.html	The release notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
machine_notes_9.30.txt	The machine notes file describes any special actions that you must take to configure and use Informix products on your computer. Machine notes are named for the product described.

♦

Windows

The following items appear in the **Informix** folder. To display this folder, choose **Start→Programs→Informix Dynamic Server 9.30→ Documentation Notes or Release Notes** from the task bar.

Program Group Item	Description
Documentation Notes	This item includes additions or corrections to manuals with information about features that might not be covered in the manuals or that have been modified since publication.
Release Notes	This item describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.

Machine notes do not apply to Windows NT platforms. ♦

- **Error message files**. Informix software products provide ASCII files that contain Informix error messages and their corrective actions. For a detailed description of these error messages, refer to *Informix Error Messages* in Answers OnLine.

UNIX

To read the error messages on UNIX, use the following command.

Command	Description
finderr	Displays error messages online

♦

WIN NT

To read error messages and corrective actions on Windows NT, use the **Informix Find Error** utility. To display this utility, choose **Start→Programs→Informix** from the task bar. ♦

Related Reading

For a list of publications that provide an introduction to database servers and operating-system platforms, refer to your *Getting Started* manual.

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

Informix Welcomes Your Comments

We want to know about any corrections or clarifications that you would find useful in our manuals that would help us with future versions. Include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Send electronic mail to us at the following address:

doc@informix.com

This address is reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact Informix Customer Services.

We appreciate your suggestions.

Chapter

1

Performance Basics

In This Chapter 1-3

A Basic Approach to Performance Measurement and Tuning 1-4

Quick Start for Small-Database User 1-5

Performance Goals 1-6

Measurements of Performance 1-7
- Throughput 1-7
  - Throughput Measurement 1-8
  - Standard Throughput Benchmarks. 1-8
- Response Time 1-10
  - Response Time and Throughput 1-11
  - Response-Time Measurement 1-12
- Cost per Transaction 1-14

Resource Utilization and Performance 1-14
- Resource Utilization 1-15
- CPU Utilization. 1-17
- Memory Utilization 1-18
- Disk Utilization. 1-20

Factors That Affect Resource Utilization 1-22

Maintenance of Good Performance 1-24

Topics Beyond the Scope of This Manual 1-26

In This Chapter

This manual discusses performance measurement and tuning for Informix Dynamic Server. Performance measurement and tuning encompass a broad area of research and practice. This manual discusses only performance tuning issues and methods that are relevant to daily database server administration and query execution. For a general introduction to performance tuning, refer to the texts listed in the "Related Reading" on page 17 in the Introduction.

For a description of the database server parallel processing and the applications that use the database server, refer to the *Getting Started with Informix Dynamic Server 2000* manual.

This chapter does the following:

- Describes a basic approach for performance measurement and tuning
- Provides guidelines for a quick start to obtain acceptable initial performance on a small database
- Describes roles in maintaining good performance
- Lists topics that are not covered in this manual

A Basic Approach to Performance Measurement and Tuning

Early indications of a performance problem are often vague; users might report that the system seems sluggish. Users might complain that they cannot get all their work done, that transactions take too long to complete, that queries take too long to process, or that the application slows down at certain times during the day. To determine the nature of the problem, you must measure the actual use of system resources and evaluate the results.

Users typically report performance problems in the following situations:

- Response times for transactions or queries take longer than expected.
- Transaction throughput is insufficient to complete the required workload.
- Transaction throughput decreases.

To maintain optimum performance for your database applications, develop a plan for measuring system performance, making adjustments to maintain good performance and taking corrective measures when performance degrades. Regular, specific measurements can help you to anticipate and correct performance problems. By recognizing problems early, you can prevent them from affecting users significantly.

Informix recommends an iterative approach to optimizing database server performance. If repeating the steps found in the following list does not produce the desired improvement, insufficient hardware resources or inefficient code in one or more client applications might be causing the problem.

To optimize performance

1. Establish performance objectives.
2. Take regular measurements of resource utilization and database activity.
3. Identify symptoms of performance problems: disproportionate utilization of CPU, memory, or disks.
4. Tune the operating-system configuration.
5. Tune the database server configuration.

6. Optimize the chunk and dbspace configuration, including placement of logs, sort space, and space for temporary tables and sort files.
7. Optimize the table placement, extent sizing, and fragmentation.
8. Improve the indexes.
9. Optimize background I/O activities, including logging, checkpoints, and page cleaning.
10. Schedule backup and batch operations for off-peak hours.
11. Optimize the implementation of the database application.
12. Repeat steps 2 through 11.

Quick Start for Small-Database User

This section is for users who have a small database with each table residing on only one disk and using only one CPU virtual processor.

To achieve acceptable initial performance on a small database

1. Generate statistics of your tables and indexes to provide information to the query optimizer to enable it to choose query plans with the lowest estimated cost.

   These statistics are a minimum starting point to obtain good performance for individual queries. For guidelines, refer to "Updating Statistics" on page 13-13. To see the query plan that the optimizer chooses for each query, refer to "Displaying the Query Plan" on page 13-6.
2. If you want a query to execute in parallel execution, you must turn on the Parallel Database Query (PDQ) feature.

   Without table fragmentation across multiple disks, parallel scans do not occur. With only one CPU virtual processor, parallel joins or parallel sorts do not occur. However, PDQ priority can obtain more memory to perform the sort. For more information, refer to Chapter 12, "Parallel Database Query."

3. If you want to mix online transaction processing (OLTP) and decision-support system (DSS) query applications, you can control the amount of resources a long-running query can obtain so that your OLTP transactions are not affected.

   For information on how to control PDQ resources, refer to "Allocating Resources for Parallel Database Queries" on page 12-12.

4. Monitor sessions and drill down into various details to improve the performance of individual queries.

   For information on the various tools and session details to monitor, refer to "Monitoring Memory Usage for Each Session" on page 13-46 and "Monitoring Sessions and Threads" on page 13-52.

Performance Goals

Many considerations go into establishing performance goals for the database server and the applications that it supports. Be clear and consistent about articulating performance goals and priorities, so that you can provide realistic and consistent expectations about the performance objectives for your application. Consider the following questions when you establish performance goals:

- Is your top priority to maximize transaction throughput, minimize response time for specific queries, or achieve the best overall mix?
- What sort of mix between simple transactions, extended decision-support queries, and other types of requests does the database server typically handle?
- At what point are you willing to trade transaction-processing speed for availability or the risk of loss for a particular transaction?
- Is this database server instance used in a client / server configuration? If so, what are the networking characteristics that affect its performance?
- What is the maximum number of users that you expect?
- Is your configuration limited by memory, disk space, or CPU resources?

The answers to these questions can help you set realistic performance goals for your resources and your mix of applications.

Measurements of Performance

The following measures describe the performance of a transaction-processing system:

- Throughput
- Response time
- Cost per transaction
- Resource utilization

Throughput, response time, and cost per transaction are described in the sections that follow. Resource utilization can have one of two meanings, depending on the context. The term can refer to the amount of a resource that a particular operation requires or uses, or it can refer to the current load on a particular system component. The term is used in the former sense to compare approaches for accomplishing a given task. For instance, if a given sort operation requires 10 megabytes of disk space, its resource utilization is greater than another sort operation that requires only 5 megabytes of disk space. The term is used in the latter sense to refer, for instance, to the number of CPU cycles that are devoted to a particular query during a specific time interval.

For a discussion of the performance impacts of different load levels on various system components, refer to "Resource Utilization and Performance" on page 1-14.

Throughput

Throughput measures the overall performance of the system. For transaction processing systems, throughput is typically measured in *transactions per second* (TPS) or *transactions per minute* (TPM). Throughput depends on the following factors:

- The specifications of the host computer
- The processing overhead in the software
- The layout of data on disk
- The degree of parallelism that both hardware and software support
- The types of transactions being processed

Throughput Measurement

The best way to measure throughput for an application is to include code in the application that logs the time stamps of transactions as they commit. If your application does not provide support for measuring throughput directly, you can obtain an estimate by tracking the number of COMMIT WORK statements that the database server logs during a given time interval. You can use the **onlog** utility to obtain a listing of logical-log records that are written to log files. You can use information from this command to track insert, delete, and update operations as well as committed transactions. However, you cannot obtain information stored in the logical-log buffer until that information is written to a log file.

If you need more immediate feedback, you can use **onstat -p** to gather an estimate. You can use the SET LOG statement to set the logging mode to unbuffered for the databases that contain tables of interest. You can also use the trusted auditing facility in the database server to record successful COMMIT WORK events or other events of interest in an audit log file. Using the auditing facility can increase the overhead involved in processing any audited event, which can reduce overall throughput. For information about the trusted auditing facility, refer to your *Trusted Facility Manual*.

Standard Throughput Benchmarks

Industrywide organizations such as the Transaction Processing Performance Council (TPC) provide standard benchmarks that allow reasonable throughput comparisons across hardware configurations and database servers. Informix is proud to be an active member in good standing of the TPC.

The TPC provides the following standardized benchmarks for measuring throughput:

- TPC-A

  This benchmark is used for simple online transaction-processing (OLTP) comparisons. It characterizes the performance of a simple transaction-processing system, emphasizing update-intensive services. TPC-A simulates a workload that consists of multiple user sessions connected over a network with significant disk I/O activity.

- TPC-B

  This benchmark is used for stress-testing peak database throughput. It uses the same transaction load as TPC-A but removes any networking and interactive operations to provide a best-case throughput measurement.

- TPC-C

  This benchmark is used for complex OLTP applications. It is derived from TPC-A and uses a mix of updates, read-only transactions, batch operations, transaction rollback requests, resource contentions, and other types of operations on a complex database to provide a better representation of typical workloads.

- TPC-D

  This benchmark measures query-processing power in terms of completion times for very large queries. TPC-D is a decision-support benchmark built around a set of typical business questions phrased as SQL queries against large databases (in the gigabyte or terabyte range).

Because every database application has its own particular workload, you cannot use TPC benchmarks to predict the throughput for your application. The actual throughput that you achieve depends largely on your application.

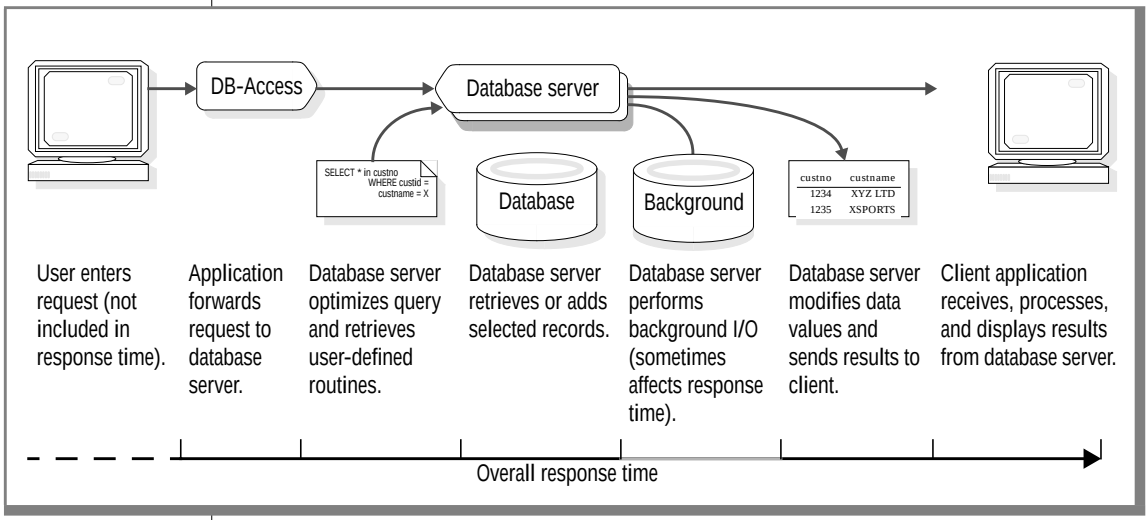
Response Time

Response time measures the performance of an individual transaction or query. Response time is typically treated as the elapsed time from the moment that a user enters a command or activates a function until the time that the application indicates the command or function has completed. The response time for a typical Dynamic Server application includes the following sequence of actions. Each action requires a certain amount of time. The response time does not include the time that it takes for the user to think of and enter a query or request:

1. The application forwards a query to the database server.
2. The database server performs query optimization and retrieves any user-defined routines (UDRs). UDRs include both SPL routines and external routines. For more information about UDRs, refer to *Extending Informix Dynamic Server*.
3. The database server retrieves, adds, or updates the appropriate records and performs disk I/O operations directly related to the query.
4. The database server performs any background I/O operations, such as logging and page cleaning, that occur during the period in which the query or transaction is still pending.
5. The database server returns a result to the application.
6. The application displays the information or issues a confirmation and then issues a new prompt to the user.

Figure 1-1 shows how these various intervals contribute to the overall response time.

Figure 1-1
Components of the Response Time for a Single Transaction


DB-Access
Database server
SELECT * in custno
WHERE custid =
custname = X
Database
Background
custno
custname
1234
XYZ LTD
1235
XSPORTS
User enters request (not included in response time).
Application forwards request to database server.
Database server optimizes query and retrieves user-defined routines.
Database server retrieves or adds selected records.
Database server performs background I/O (sometimes affects response time).
Database server modifies data values and sends results to client.
Client application receives, processes, and displays results from database server.
Overall response time

Response Time and Throughput

Response time and throughput are related. The response time for an average transaction tends to decrease as you increase overall throughput. However, you can decrease the response time for a specific query, at the expense of overall throughput, by allocating a disproportionate amount of resources to that query. Conversely, you can maintain overall throughput by restricting the resources that the database allocates to a large query.

The trade-off between throughput and response time becomes evident when you try to balance the ongoing need for high transaction throughput with an immediate need to perform a large decision-support query. The more resources that you apply to the query, the fewer you have available to process transactions, and the larger the impact your query can have on transaction throughput. Conversely, the fewer resources you allow the query, the longer the query takes.

Response-Time Measurement

You can use either of the following methods to measure response time for a query or application:

- Operating-system timing commands
- Operating-system performance monitor
- Timing functions within your application

Operating-System Timing Commands

Your operating system typically has a utility that you can use to time a command. You can often use this timing utility to measure the response times to SQL statements that a DB-Access command file issues.

UNIX

If you have a command file that performs a standard set of SQL statements, you can use the **time** command on many systems to obtain an accurate timing for those commands. For more information about command files, refer to the *DB-Access User's Manual*. The following example shows the output of the UNIX **time** command:

```
time commands.dba
...
4.3 real 1.5 user     1.3 sys
```

The **time** output lists the amount of elapsed time (real), the user CPU time, and the system CPU time. If you use the C shell, the first three columns of output from the C shell **time** command show the user, system, and elapsed times, respectively. In general, an application often performs poorly when the proportion of system CPU time exceeds one-third of the total elapsed time.

The **time** command gathers timing information about your application. You can use this command to invoke an instance of your application, perform a database operation, and then exit to obtain timing figures, as the following example illustrates:

```
time sqlapp
   (enter SQL command through sqlapp, then exit)
10.1 real 6.4 user 3.7 sys
```

You can use a script to run the same test repeatedly, which allows you to obtain comparable results under different conditions. You can also obtain estimates of your average response time by dividing the elapsed time for the script by the number of database operations that the script performs. ♦

Operating-System Performance Monitor

Operating systems usually have a performance monitor that you can use to measure response time for a query or process.

WIN NT

You can often use the Performance Monitor that the Windows NT operating system supplies to measure the following times:

- User time
- Processor time
- Elapsed time ♦

Timing Functions Within Your Application

Most programming languages have a library function for the time of day. If you have access to the source code, you can insert pairs of calls to this function to measure the elapsed time between specific actions.

E/C

For example, if the application is written in Informix ESQL/C, you can use the **dtcurrent()** function to obtain the current time. To measure response time, you can call **dtcurrent()** to report the time at the start of a transaction and again to report the time when the transaction commits. ♦

Elapsed time, in a multiprogramming system or network environment where resources are shared among multiple processes, does not always correspond to execution time. Most operating systems and C libraries contain functions that return the CPU time of a program.

Cost per Transaction

The cost per transaction is a financial measure that is typically used to compare overall operating costs among applications, database servers, or hardware platforms.

To measure the cost per transaction

1. Calculate all the costs associated with operating an application.

   These costs can include the installed price of the hardware and software; operating costs, including salaries; and other expenses.
2. Project the total number of transactions and queries for the effective life of an application.
3. Divide the total cost over the total number of transactions.

Although this measure is useful for planning and evaluation, it is seldom relevant to the daily issues of achieving optimum performance.

Resource Utilization and Performance

A typical transaction-processing application undergoes different demands throughout its various operating cycles. Peak loads during the day, week, month, and year, as well as the loads imposed by decision-support (DSS) queries or backup operations, can have significant impact on any system that is running near capacity. You can use direct historical data derived from your particular system to pinpoint this impact.

You must take regular measurements of the workload and performance of your system to predict peak loads and compare performance measurements at different points in your usage cycle. Regular measurements help you to develop an overall performance profile for your database server applications. This profile is critical in determining how to improve performance reliably.

For the measurement tools that the database server provides, refer to "Database Server Tools" on page 2-6. For the tools that your operating system provides for measuring performance impacts on system and hardware resources, refer to "Operating-System Tools" on page 2-5.

Utilization is the percentage of time that a component is actually occupied, as compared with the total time that the component is available for use. For instance, if a CPU processes transactions for a total of 40 seconds during a single minute, its utilization during that interval is 67 percent.

Measure and record utilization of the following system resources regularly:

- CPU
- Memory
- Disk

A resource is said to be *critical* to performance when it becomes overused or when its utilization is disproportionate to that of other components. For instance, you might consider a disk to be critical or overused when it has a utilization of 70 percent and all other disks on the system have 30 percent. Although 70 percent does not indicate that the disk is severely overused, you could improve performance by rearranging data to balance I/O requests across the entire set of disks.

How you measure resource utilization depends on the tools that your operating system provides for reporting system activity and resource utilization. Once you identify a resource that seems overused, you can use database server performance-monitoring utilities to gather data and make inferences about the database activities that might account for the load on that component. You can adjust your database server configuration or your operating system to reduce those database activities or spread them among other components. In some cases, you might need to provide additional hardware resources to resolve a performance bottleneck.

Resource Utilization

Whenever a system resource, such as a CPU or a particular disk, is occupied by a transaction or query, it is unavailable for processing other requests. Pending requests must wait for the resources to become available before they can complete. When a component is too busy to keep up with all its requests, the overused component becomes a bottleneck in the flow of activity. The higher the percentage of time that the resource is occupied, the longer each operation must wait for its turn.

You can use the following formula to estimate the service time for a request based on the overall utilization of the component that services the request. The expected service time includes the time that is spent both waiting for and using the resource in question. Think of service time as that portion of the response time accounted for by a single component within your computer, as the following formula shows:

$$S \approx P/(1-U)$$

S is the expected service time.

P is the processing time that the operation requires once it obtains the resource.

U is the utilization for the resource (expressed as a decimal).

As Figure 1-2 shows, the service time for a single component increases dramatically as the utilization increases beyond 70 percent. For instance, if a transaction requires 1 second of processing by a given component, you can expect it to take 2 seconds on a component at 50 percent utilization and 5 seconds on a component at 80 percent utilization. When utilization for the resource reaches 90 percent, you can expect the transaction to take 10 seconds to make its way through that component.

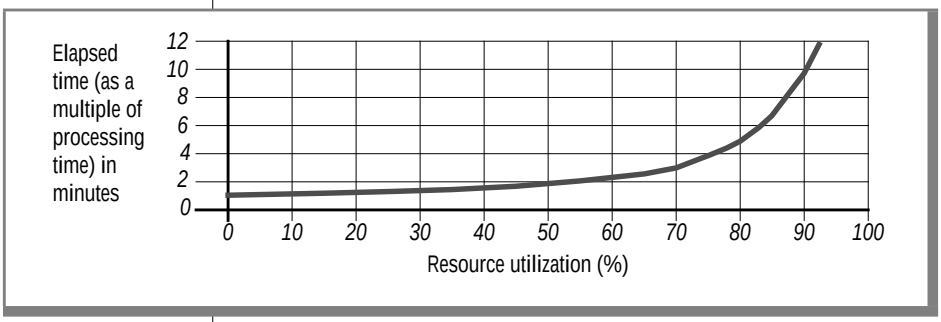


Figure 1-2
Service Time for a Single Component as a Function of Resource Utilization

If the average response time for a typical transaction soars from 2 or 3 seconds to 10 seconds or more, users are certain to notice and complain.

Important: *Monitor any system resource that shows a utilization of over 70 percent or any resource that exhibits symptoms of overuse as described in the following sections.*

CPU Utilization

You can use the resource-utilization formula from the previous section to estimate the response time for a heavily loaded CPU. However, high utilization for the CPU does not always indicate a performance problem. The CPU performs all calculations that are needed to process transactions. The more transaction-related calculations that it performs within a given period, the higher the throughput will be for that period. As long as transaction throughput is high and seems to remain proportional to CPU utilization, a high CPU utilization indicates that the computer is being used to the fullest advantage.

On the other hand, when CPU utilization is high but transaction throughput does not keep pace, the CPU is either processing transactions inefficiently or it is engaged in activity not directly related to transaction processing. CPU cycles are being diverted to internal housekeeping tasks such as memory management. You can easily eliminate the following activities:

- Large queries that might be better scheduled at an off-peak time
- Unrelated application programs that might be better performed on another computer

If the response time for transactions increases to such an extent that delays become unacceptable, the processor might be swamped; the transaction load might be too high for the computer to manage. Slow response time can also indicate that the CPU is processing transactions inefficiently or that CPU cycles are being diverted.

When CPU utilization is high, a detailed analysis of the activities that the database server performs can reveal any sources of inefficiency that might be present due to improper configuration. For information about analyzing database server activity, refer to "Database Server Tools" on page 2-6.

Memory Utilization

Although the principle for estimating the service time for memory is the same as that described in "Resource Utilization and Performance" on page 1-14, you use a different formula to estimate the performance impact of memory utilization than you do for other system components. Memory is not managed as a single component such as a CPU or disk, but as a collection of small components called *pages*. The size of a typical page in memory can range from 1 to 8 kilobytes, depending on your operating system. A computer with 64 megabytes of memory and a page size of 2 kilobytes contains approximately 32,000 pages.

When the operating system needs to allocate memory for use by a process, it scavenges any unused pages within memory that it can find. If no free pages exist, the memory-management system has to choose pages that other processes are still using and that seem least likely to be needed in the short run. CPU cycles are required to select those pages. The process of locating such pages is called a *page scan*. CPU utilization increases when a page scan is required.

Memory-management systems typically use a *least recently used* algorithm to select pages that can be copied out to disk and then freed for use by other processes. When the CPU has identified pages that it can appropriate, it *pages out* the old page images by copying the old data from those pages to a dedicated disk. The disk or disk partition that stores the page images is called the *swap disk*, *swap space*, or *swap area*. This paging activity requires CPU cycles as well as I/O operations.

Eventually, page images that have been copied to the swap disk must be brought back in for use by the processes that require them. If there are still too few free pages, more must be paged out to make room. As memory comes under increasing demand and paging activity increases, this activity can reach a point at which the CPU is almost fully occupied with paging activity. A system in this condition is said to be *thrashing*. When a computer is thrashing, all useful work comes to a halt.

To prevent thrashing, some operating systems use a coarser memory-management algorithm after paging activity crosses a certain threshold. This algorithm is called *swapping*. When the memory-management system resorts to swapping, it appropriates all pages that constitute an entire process image at once, rather than a page at a time. Swapping frees up more memory with each operation. However, as swapping continues, every process that is swapped out must be read in again, dramatically increasing disk I/O to the swap device and the time required to switch between processes. Performance is then limited to the speed at which data can be transferred from the swap disk back into memory. Swapping is a symptom of a system that is severely overloaded, and throughput is impaired.

Many systems provide information about paging activity that includes the number of page scans performed, the number of pages sent out of memory (*paged out*), and the number of pages brought in from memory (*paged in)*:

- Paging out is the critical factor because the operating system pages out only when it cannot find pages that are free already.
- A high rate of page scans provides an early indicator that memory utilization is becoming a bottleneck.
- Pages for terminated processes are freed in place and simply reused, so paging-in activity does not provide an accurate reflection of the load on memory. A high rate of paging in can result from a high rate of process turnover with no significant performance impact.

You can use the following formula to calculate the expected paging delay for a given CPU utilization level and paging rate:

$$PD \approx (C/(1-U)) * R * T$$

PD is the paging delay.

C is the CPU service time for a transaction.

U is the CPU utilization (expressed as a decimal).

R is the paging-out rate.

T is the service time for the swap device.

As paging increases, CPU utilization also increases, and these increases are compounded. If a paging rate of 10 per second accounts for 5 percent of CPU utilization, increasing the paging rate to 20 per second might increase CPU utilization by an additional 5 percent. Further increases in paging lead to even sharper increases in CPU utilization, until the expected service time for CPU requests becomes unacceptable.

Disk Utilization

Because each disk acts as a single resource, you can use the following basic formula to estimate the service time, which is described in detail in "Resource Utilization" on page 1-15:

$$S \approx P/(1-U)$$

However, because transfer rates vary among disks, most operating systems do not report disk utilization directly. Instead, they report the number of data transfers per second (in operating-system memory-page-size units.) To compare the load on disks with similar access times, simply compare the average number of transfers per second.

If you know the access time for a given disk, you can use the number of transfers per second that the operating system reports to calculate utilization for the disk. To do so, multiply the average number of transfers per second by the access time for the disk as listed by the disk manufacturer. Depending on how your data is laid out on the disk, your access times can vary from the rating of the manufacturer. To account for this variability, Informix recommends that you add 20 percent to the access-time specification of the manufacturer.

The following example shows how to calculate the utilization for a disk with a 30-millisecond access time and an average of 10 transfer requests per second:

```
U = (A * 1.2) * X
  = (.03 * 1.2) * 10
  = .36
```

U is the resource utilization (this time of a disk).

A is the access time (in seconds) that the manufacturer lists.

X is the number of transfers per second that your operating system reports.

You can use the utilization to estimate the processing time at the disk for a transaction that requires a given number of disk transfers. To calculate the processing time at the disk, multiply the number of disk transfers by the average access time. Include an extra 20 percent to account for access-time variability:

```
P = D (A * 1.2)
```

P is the processing time at the disk.

D is the number of disk transfers.

A is the access time (in seconds) that the manufacturer lists.

For example, you can calculate the processing time for a transaction that requires 20 disk transfers from a 30-millisecond disk as follows:

```
P = 20 (.03 * 1.2)
  = 20 * .036
  = .72
```

Use the processing time and utilization values that you calculated to estimate the expected service time for I/O at the particular disk, as the following example shows:

```
S ≈ P/(1-U)
  = .72 / (1 - .36)
  = .72 / .64
  = 1.13
```

Factors That Affect Resource Utilization

The performance of your database server application depends on the following factors. You must consider these factors when you attempt to identify performance problems or make adjustments to your system:

- Hardware resources

  As discussed earlier in this chapter, hardware resources include the CPU, physical memory, and disk I/O subsystems.

- Operating-system configuration

  The database server depends on the operating system to provide low-level access to devices, process scheduling, interprocess communication, and other vital services.

  The configuration of your operating system has a direct impact on how well the database server performs. The operating-system kernel takes up a significant amount of physical memory that the database server or other applications cannot use. However, you must reserve adequate kernel resources for the database server to use.

- Network configuration and traffic

  Applications that depend on a network for communication with the database server, and systems that rely on data replication to maintain high availability, are subject to the performance constraints of that network. Data transfers over a network are typically slower than data transfers from a disk. Network delays can have a significant impact on the performance of the database server and other application programs that run on the host computer.

- Database server configuration

  Characteristics of your database server instance, such as the number of CPU virtual processors (VPs), the size of your resident and virtual shared-memory portions, and the number of users, play an important role in determining the capacity and performance of your applications.

- Dbspace, blobspace, and chunk configuration

  The following factors can affect the time that it takes the database server to perform disk I/O and process transactions:

  - The placement of the root dbspace, physical logs, logical logs, and temporary-table dbspaces
  - The presence or absence of mirroring
  - The use of devices that are buffered or unbuffered by the operation system

- Database and table placement

  The placement of tables and fragments within dbspaces, the isolation of high-use fragments in separate dbspaces, and the spreading of fragments across multiple dbspaces can affect the speed at which the database server can locate data pages and transfer them to memory.

- Tblspace organization and extent sizing

  Fragmentation strategy and the size and placement of extents can affect the ability of the database server to scan a table rapidly for data. Avoid interleaved extents and allocate extents that are sufficient to accommodate growth of a table to prevent performance problems.

- Query efficiency

  Proper query construction and cursor use can decrease the load that any one application or user imposes. Remind users and application developers that others require access to the database and that each person's activities affect the resources that are available to others.

- Scheduling background I/O activities

  Logging, checkpoints, page cleaning, and other operations, such as making backups or running large decision-support queries, can impose constant overhead and large temporary loads on the system. Schedule backup and batch operations for off-peak times whenever possible.

- Remote client / server operations and distributed join operations

  These operations have an important impact on performance, especially on a host system that coordinates distributed joins.

- Application-code efficiency

  Application programs introduce their own load on the operating system, the network, and the database server. These programs can introduce performance problems if they make poor use of system resources, generate undue network traffic, or create unnecessary contention in the database server. Application developers must make proper use of cursors and locking levels to ensure good database server performance.

Maintenance of Good Performance

Performance is affected in some way by all system users: the database server administrator, the database administrator, the application designers, and the client application users.

The database server administrator usually coordinates the activities of all users to ensure that system performance meets overall expectations. For example, the operating-system administrator might need to reconfigure the operating system to increase the amount of shared memory. Bringing down the operating system to install the new configuration requires bringing the database server down. The database server administrator must schedule this downtime and notify all affected users when the system will be unavailable.

The database server administrator should:

- Be aware of all performance-related activities that occur.
- Educate users about the importance of performance, how performance-related activities affect them, and how they can assist in achieving and maintaining optimal performance.

The database administrator should pay attention to:

- How tables and queries affect the overall performance of the database server
- The placement of tables and fragments
- How the distribution of data across disks affects performance

Application developers should:

- Carefully design applications to use the concurrency and sorting facilities that the database server provides, rather than attempt to implement similar facilities in the application.
- Keep the scope and duration of locks to the minimum to avoid contention for database resources.
- Include routines within applications that, when temporarily enabled at runtime, allow the database server administrator to monitor response times and transaction throughput.

Database users should:

- Pay attention to performance and report problems to the database server administrator promptly.
- Be courteous when they schedule large, decision-support queries and request as few resources as possible to get the work done.

Topics Beyond the Scope of This Manual

Attempts to balance the workload often produce a succession of moderate performance improvements. Sometimes the improvements are dramatic. However, in some situations a load-balancing approach is not enough. The following types of situations might require measures beyond the scope of this manual:

- Application programs that require modification to make better use of database server or operating-system resources
- Applications that interact in ways that impair performance
- A host computer that might be subject to conflicting uses
- A host computer with capacity that is inadequate for the evolving workload
- Network performance problems that affect client/server or other applications

No amount of database tuning can correct these situations. Nevertheless, they are easier to identify and resolve when the database server is configured properly.

Important: *Although broad performance considerations also include reliability and data availability as well as improved response time and efficient use of system resources, this manual discusses only response time and system resource use. For discussions of improved database server reliability and data availability, see information about switchover, mirroring, and high availability in your "Administrator's Guide." For information about backup and restore, see the "Backup and Restore Guide."*

Chapter

2

Performance Monitoring

In This Chapter 2-3

Evaluating the Current Configuration 2-3

Creating a Performance History 2-4
- The Importance of a Performance History 2-4
- Tools That Create a Performance History 2-5
  - Operating-System Tools 2-5
  - Database Server Tools 2-6

Monitoring Database Server Resources 2-11
- Monitoring Resources That Impact CPU Utilization 2-11
- Monitoring Memory Utilization 2-13
- Monitoring Disk I/O Utilization 2-14
  - Using onstat -g to Monitor I/O Utilization 2-15
  - Using ISA to Monitor I/O Utilization 2-15
  - Using the oncheck Utility to Monitor I/O Utilization 2-16

Monitoring Transactions 2-18
- The onlog Utility 2-18
- Using the onstat Utility to Monitor Transactions 2-19
- Using ISA to Monitor Transactions 2-19

Monitoring Sessions and Queries 2-20
- Monitoring Memory Usage for Each Session. 2-20
- Using SET EXPLAIN 2-20

In This Chapter

This chapter describes the performance monitoring tools and cross-references sections in subsequent chapters for how to interpret the results of performance monitoring. The descriptions of the tools can help you decide which tools to use to create a performance history, to monitor database resources at scheduled times, or to monitor ongoing transaction or query performance.

The kinds of data that you need to collect depend on the kinds of applications that you run on your system. The causes of performance problems on OLTP (online transaction processing) systems are different from the causes of problems on systems that are used primarily for DSS query applications. Systems with mixed use provide a greater performance-tuning challenge and require a sophisticated analysis of performance-problem causes.

Evaluating the Current Configuration

Before you begin to adjust the configuration of your database server, evaluate the performance of your current configuration. To alter certain database server characteristics, you must bring down the database server, which can affect your production system. Some configuration adjustments can unintentionally decrease performance or cause other negative side effects.

If your database applications perform well enough to satisfy user expectations, you should avoid frequent adjustments, even if those adjustments might theoretically improve performance. As long as your users are reasonably satisfied, take a measured approach to reconfiguring the database server. When possible, use a test instance of the database server to evaluate configuration changes before you reconfigure your production system.

To examine your current database server configuration, you can use the utilities described in this chapter.

When performance problems relate to backup operations, you might also examine the number or transfer rates for tape drives. You might need to alter the layout or fragmentation of your tables to reduce the impact of backup operations. For information about disk layout and table fragmentation, refer to Chapter 6, "Table Performance Considerations," and Chapter 7, "Index Performance Considerations."

For client/server configurations, consider network performance and availability. Evaluating network performance is beyond the scope of this manual. For information on monitoring network activity and improving network availability, see your network administrator or refer to the documentation for your networking package.

Creating a Performance History

As soon as you set up your database server and begin to run applications on it, you should begin scheduled monitoring of resource use. To accumulate data for performance analysis, use the command-line utilities described in "Database Server Tools" on page 2-6 and "Operating-System Tools" on page 2-5 in operating scripts or batch files.

The Importance of a Performance History

To build a performance history and profile of your system, take regular snapshots of resource-utilization information. For example, if you chart the CPU utilization, paging-out rate, and the I/O transfer rates for the various disks on your system, you can begin to identify peak-use levels, peak-use intervals, and heavily loaded resources. If you monitor fragment use, you can determine whether your fragmentation scheme is correctly configured. Monitor other resource use as appropriate for your database server configuration and the applications that run on it.

If you have this information on hand, you can begin to track the cause of problems as soon as users report slow response or inadequate throughput. If history is not available, you must start tracking performance after a problem arises, and you might not be able to tell when and how the problem began. Trying to identify problems after the fact significantly delays resolution of a performance problem.

Choose tools from those described in the following sections, and create jobs that build up a history of disk, memory, I/O, and other database server resource use. To help you decide which tools to use to create a performance history, this chapter briefly describes the output of each tool.

Tools That Create a Performance History

When you monitor database server performance, you use tools from the host operating system and command-line utilities that you can run at regular intervals from scripts or batch files. You also use performance monitoring tools with a graphical interface to monitor critical aspects of performance as queries and transactions are performed.

Operating-System Tools

The database server relies on the operating system of the host computer to provide access to system resources such as the CPU, memory, and various unbuffered disk I/O interfaces and files. Each operating system has its own set of utilities for reporting how system resources are used. Different implementations of some operating systems have monitoring utilities with the same name but different options and informational displays.

UNIX

You might be able to use some of the following typical UNIX operating-system resource-monitor utilities.

UNIX Utility	Description
vmstat	Displays virtual-memory statistics
iostat	Displays I/O utilization statistics
sar	Displays a variety of resource statistics
ps	Displays active process information

For details on how to monitor your operating-system resources, consult the reference manual or your system administration guide.

To capture the status of system resources at regular intervals, use scheduling tools that are available with your host operating system (for example, **cron**) as part of your performance monitoring system. ♦

WIN NT

Windows NT supplies a Performance Monitor (**perfmon.exe**) that can monitor resources such as processor, memory, cache, threads, and processes. The Performance Monitor also provides charts, alerts, reports, and the ability to save information to log files for later analysis.

For more information on how to use the Performance Monitor, consult your operating-system manuals. ♦

Database Server Tools

The database server provides utilities to capture snapshot information about your configuration and performance. It also provides the system-monitoring interface (SMI) for monitoring performance from within your application.

You can use these utilities regularly to build a historical profile of database activity, which you can compare with current operating-system resource-utilization data. These comparisons can help you discover which database server activities have the greatest impact on system-resource utilization. You can use this information to identify and manage your high-impact activities or adjust your database server or operating-system configuration.

The database server provides the following utilities:

- Informix Server Administrator (ISA)
- **onstat**
- **onlog**
- **oncheck**
- DB-Access and the system-monitoring interface (SMI)
- ON-Monitor
- **onperf** ♦

UNIX

You can use **onstat**, **onlog**, or **oncheck** commands invoked by the **cron** scheduling facility to capture performance-related information at regular intervals and build a historical performance profile of your database server application. The following sections describe these utilities.

You can use SQL SELECT statements to query the system-monitoring interface (SMI) from within your application.

The SMI tables are a collection of tables and pseudo-tables in the **sysmaster** database that contain dynamically updated information about the operation of the database server. The database server constructs these tables in memory but does not record them on disk. The **onstat** utility options obtain information from these SMI tables.

You can use **cron** and SQL scripts with DB-Access or **onstat** utility options to query SMI tables at regular intervals. For information about SQL scripts, refer to the *DB-Access User's Manual*. For information about SMI tables, refer to your *Administrator's Reference*.

Tip: *The SMI tables are different from the system catalog tables. System catalog tables contain permanently stored and updated information about each database and its tables (sometimes referred to as "metadata" or a "data dictionary"). For information about SMI tables, refer to the Administrator's Reference. For information about system catalog tables, refer to the Informix Guide to SQL: Reference.*

You can use ON-Monitor to check the current database server configuration. For information about ON-Monitor, refer to your *Administrator's Reference*.

You can use **onperf** to display database server activity with the Motif window manager. For information about **onperf**, refer to Chapter 14, "The onperf Utility on UNIX." ♦

Informix Server Administrator

Informix Server Administrator (ISA) is a browser-based tool that provides Web-based system administration for the entire range of Informix database servers. ISA is the first in a new generation of browser-based, cross-platform administrative tools. It provides access to every Informix database server command-line function and presents the output in an easy-to-read format.

The database server CD-ROM distributed with your product includes ISA. For information on how to install ISA, see the following file on the CD-ROM.

Operating System	File
UNIX	**/SVR_ADM/README**
Windows NT	**\SVR_ADM\readme.txt**

With ISA, you can use a browser to perform these common database server administrative tasks:

- Change configuration parameters temporarily or permanently
- Change the database server mode between online and offline and its intermediate states
- Modify connectivity information in the **sqlhosts** file
- Check dbspaces, sbspaces, logs, and other objects
- Manage logical and physical logs
- Examine memory use and adding and freeing memory segments
- Read the message log
- Back up and restore dbspaces and sbspaces
- Run various **onstat** commands to monitor performance
- Enter simple SQL statements and examine database schemas
- Add and remove chunks, dbspaces, and sbspaces
- Examine and manage user sessions
- Examine and manage virtual processors (VPs)
- Use the High-Performance Loader (HPL), **dbimport**, and **dbexport**

- Manage Enterprise Replication
- Manage an Informix MaxConnect server
- Use the following utilities: **dbaccess**, **dbschema**, **onbar**, **oncheck**, **ondblog**, **oninit**, **onlog**, **onmode**, **onparams**, **onspaces**, and **onstat**

You also can enter any Informix utility, UNIX shell command, or Windows NT command (for example, **oncheck -cd**; **ls -l**).

The onstat Utility

You can use the **onstat** utility to check the current status of the database server and monitor the activities of the database server. This utility displays a wide variety of performance-related and status information contained within the SMI tables. For a complete list of all **onstat** options, use **onstat - -**. For a complete display of all the information that **onstat** gathers, use **onstat-a**.

Tip: *Profile information displayed by **onstat** commands, such as **onstat -p**, accumulates from the time the database server was initialized. To clear performance profile statistics so that you can create a new profile, run **onstat -z**. If you use **onstat -z** to reset statistics for a performance history or appraisal, ensure that other users do not also enter the command at different intervals.*

The following table lists **onstat** options that display general performance-related information.

Option	Description
onstat -p	Displays a performance profile that includes the number of reads and writes, the number of times that a resource was requested but was not available, and other miscellaneous information
onstat -b	Displays information about buffers currently in use
onstat -l	Displays information about the physical and logical logs
onstat -x	Displays information about transactions, including the thread identifier of the user who owns the transaction
onstat -u	Displays a user activity profile that provides information about user threads including the thread owner's session ID and login name
onstat -R	Displays each least recently used (LRU) queue, type, and the total number of queued and dirty buffers For the performance implications of LRU queue statistics, see "LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY" on page 5-56.
onstat -F	Displays page-cleaning statistics that include the number of writes of each type that flushes pages to disk
onstat -g	Requires an additional argument that specifies the information to be displayed, such as **onstat -g mem** to display memory statistics

For more information about options that provide performance-related information, refer to "Monitoring Database Server Resources" on page 2-11. For a list and explanation of **onstat -g** arguments, refer to your *Administrator's Reference*.

Monitoring Database Server Resources

Monitor specific database server resources to identify performance bottlenecks and potential trouble spots and improve resource use and response time.

One of the most useful commands for monitoring system resources is **onstat -g** and its many options. ISA executes **onstat** options to display information. "Monitoring Fragment Use" on page 9-39 and "Monitoring and Tuning the SQL Statement Cache" on page 4-42 contain many **onstat -g** examples.

Monitoring Resources That Impact CPU Utilization

The following database server resources impact CPU utilization:

- Threads
- Network communications
- Virtual processors

Use the following **onstat -g** arguments to monitor threads.

Argument	Description
act	Displays active threads
ath	Displays all threads The sqlexec threads represent portions of client sessions; the rstcb value corresponds to the user field of the **onstat -u** command.
rea	Displays ready threads
sle	Displays all sleeping threads

(1 of 2)

Argument	Description
sts	Displays maximum and current stack use per thread
tpf *tid*	Displays a thread profile for *tid* If *tid* is 0, this argument displays profiles for all threads.
wai	Displays waiting threads, including all threads waiting on mutex or condition, or yielding

(2 of 2)

Use the following **onstat -g** arguments to monitor the network.

Argument	Description
ntd	Displays network statistics by service
ntt	Displays network user times
ntu	Displays network user statistics
qst	Displays queue statistics

Use the following **onstat -g** arguments to monitor virtual processors.

Argument	Description
glo	Displays global multithreading information, including CPU-use information about virtual processors, the total number of sessions, and other multithreading global counters
sch	Displays the number of semaphore operations, spins, and busy waits for each VP
spi	Displays longspins, which are spin locks that virtual processors have spun more than 10,000 times in order to acquire To reduce longspins, reduce the number of virtual processors, reduce the load on the computer, or, on some platforms, use the *no-age* or *processor affinity* features.
wst	Displays wait statistics

Monitoring Memory Utilization

Use the following **onstat -g** arguments to monitor memory utilization. For overall memory information, omit *table name*, *pool name*, or *session id* from the commands that permit those optional parameters.

Argument	Description
ffr *pool name* | *session id*	Displays free fragments for a pool of shared memory or by session
dic *table name*	Displays one line of information for each table cached in the shared-memory dictionary If you provide a specific table name as a parameter, this argument displays internal SQL information about that table.
dsc	Displays one line of information for each column of distribution statistics cached in the data distribution cache.
mem *pool name* | *session id*	Displays memory statistics for the pools that are associated with a session If you omit *pool_name* | *session id*, this argument displays pool information for all sessions.
mgm	Displays memory grant manager resource information
nsc *client id*	Displays shared-memory status by client ID If you omit *client id*, this argument displays all client status areas.
nsd	Displays network shared-memory data for poll threads
nss *session id*	Displays network shared-memory status by session id If you omit session id, this argument displays all session status areas.
prc	Displays one line of information for each user-defined routine (SPL routine or external routine written in C or Java programming language) cached in the UDR cache
seg	Displays shared-memory-segment statistics This argument shows the number and size of all attached segments.

(1 of 2)

Argument	Description
ses *session id*	Displays memory usage for session id If you omit session id, this argument displays memory usage for all sessions.
ssc	Displays one line of information for each query cached in the SQL statement cache
stm *session id*	Displays memory usage of each SQL statement for session id If you omit session id, this argument displays memory usage for all sessions.
ufr *pool name* | *session id*	Displays allocated pool fragments by user or session

(2 of 2)

Monitoring Disk I/O Utilization

Use any of the following utilities to determine if your disk I/O operations are efficient for your applications:

- **onstat -g** arguments
- ISA
- **oncheck** utility

Using onstat -g to Monitor I/O Utilization

Use the following **onstat -g** arguments to monitor disk I/O utilization.

Argument	Description
iof	Displays asynchronous I/O statistics by chunk or file This argument is similar to the **onstat -d**, except that information on nonchunk files also appears. This argument displays information about temporary dbspaces and sort files.
iog	Displays asynchronous I/O global information
ioq	Displays asynchronous I/O queuing statistics
iov	Displays asynchronous I/O statistics by virtual processor

For a detailed case study that uses various **onstat** outputs, refer to "Case Studies and Examples" on page A-1.

Using ISA to Monitor I/O Utilization

You can use ISA to monitor disk I/O utilization. ISA uses information that the following **onstat** command-line options generate to display session information, as the following table shows.

To monitor	Select on ISA	Displays onstat output
Asynchronous I/O statistics by chunk or file	**Performance→AIO→ Disk I/O by Queue**	**onstat -g iof**
Asynchronous I/O global information	**Performance→AIO→ Global Information**	**onstat -g iog**
Asynchronous I/O statistics by virtual processor	**Performance→AIO→ Disk I/O by VP**	**onstat -g iov**
Asynchronous I/O queuing statistics	**Performance→AIO→ Disk I/O by File**	**onstat -g ioq**

Using the oncheck Utility to Monitor I/O Utilization

Disk I/O operations are usually the longest component of the response time for a query. Contiguously allocated disk space improves sequential disk I/O operations because the database server can read in larger blocks of data and use the read-ahead feature to reduce the number of I/O operations.

The **oncheck** utility displays information about storage structures on a disk, including chunks, dbspaces, blobspaces, extents, data rows, system catalog tables, and other options. You can also use **oncheck** to determine the number of extents that exist within a table and whether or not a table occupies contiguous space.

The **oncheck** utility provides the following options and information that apply to contiguous space and extents. For information on how to use other **oncheck** options, refer to the *Administrator's Guide for Informix Dynamic Server*.

Option	Information
-pB	Blobspace simple large object (TEXT or BYTE data) For information on how to use this option to determine the efficiency of blobpage size, refer to "Determining Blobpage Fullness with oncheck -pB" on page 5-24.
-pe	Chunks and extents For information on how to use this option to monitor extents, refer to "Checking for Extent Interleaving" on page 6-41 and "Eliminating Interleaved Extents" on page 6-42.
-pk	Index key values. For information on how to improve the performance of this option, refer to"Improving Performance for Index Checks" on page 7-21.
-pK	Index keys and row IDs For information on how to improve the performance of this option, refer to"Improving Performance for Index Checks" on page 7-21.
-pl	Index-leaf key values For information on how to improve the performance of this option, refer to"Improving Performance for Index Checks" on page 7-21.

(1 of 2)

Option	Information
-pL	Index-leaf key values and row IDs For information on how to improve the performance of this option, refer to"Improving Performance for Index Checks" on page 7-21.
-pp	Pages by table or fragment For information on how to use this option to monitor space, refer to "Considering the Upper Limit on Extents" on page 6-39.
-pP	Pages by chunk For information on how to use this option to monitor extents, refer to "Considering the Upper Limit on Extents" on page 6-39.
-pr	Root reserved pages For information on how to use this option, refer to "Estimating Tables with Fixed-Length Rows" on page 6-12.
-ps	Space used by smart large objects and metadata in sbspace
-pS	Space used by smart large objects and metadata in sbspace and storage characteristics For information on how to use this option to monitor space, refer to "Monitoring Sbspaces" on page 6-25.
-pt	Space used by table or fragment For information on how to use this option to monitor space, refer to "Estimating Table Size" on page 6-12.
-pT	Space used by table, including indexes For information on how to use this option to monitor space, refer to "Performance Considerations for DDL Statements" on page 6-57.

(2 of 2)

For more information about using **oncheck** to monitor space, refer to "Estimating Table Size" on page 6-12. For more information on concurrency during **oncheck** execution, refer to "Improving Performance for Index Checks" on page 7-21. For more **oncheck** information, refer to your *Administrator's Reference*.

Monitoring Transactions

Use any of the following utilities to monitor transactions:

- **onlog** utility
- **onstat** utility
- ISA

The onlog Utility

The **onlog** utility displays all or selected portions of the logical log. This command can take input from selected log files, the entire logical log, or a backup tape of previous log files. The **onlog** utility can help you to identify a problematic transaction or to gauge transaction activity that corresponds to a period of high utilization, as indicated by your periodic snapshots of database activity and system-resource consumption.

Use **onlog** with caution when you read logical-log files still on disk, because attempting to read unreleased log files stops other database activity. For greatest safety, Informix recommends that you back up the logical-log files first and then read the contents of the backup files. With proper care, you can use the **onlog -n** option to restrict **onlog** only to logical-log files that have been released. To check on the status of logical-log files, use **onstat -l**. For more information about **onlog**, refer to your *Administrator's Reference*.

Using the onstat Utility to Monitor Transactions

If the throughput of transactions is not very high, you can use the following **onstat** options to identify which transaction might be a bottleneck.

Option	Description
onstat -x	Displays transaction information such as number of locks held and isolation level.
onstat -u	Displays information about each user thread
onstat -k	Displays locks held by each session
onstat -g sql	Displays last SQL statement this session executed

Using ISA to Monitor Transactions

Monitor transactions to track open transactions and the locks that those transactions hold. You can use ISA to monitor transactions and user sessions. ISA uses information that the following **onstat** command-line options generate to display session information, as the following table shows. Click the **Refresh** button to rerun the **onstat** command and display fresh information.

To monitor	Select on ISA	Displays onstat output	Refer to
Transaction statistics, such as number of locks held and isolation level	**Users→Transaction**	**onstat -x**	"Displaying Transactions with onstat -x" on page 13-59
User session statistics	**Users→Threads**	**onstat -u**	"Displaying User Sessions with onstat -u" on page 13-62
Lock statistics	**Performance→Locks**	**onstat -k**	"Displaying Locks with onstat -k" on page 13-61
Sessions running SQL statements	**Users→Connections→ session-id**	**onstat -g sql** *sessid*	"Displaying Sessions Executing SQL Statements" on page 13-63

Monitoring Sessions and Queries

To monitor database server activity, you can view the number of active sessions and the amount of resources that they are using. Monitoring sessions and threads is important for sessions that perform queries as well as sessions that perform inserts, updates, and deletes. Some of the information that you can monitor for sessions and threads allows you to determine if an application is using a disproportionate amount of the resources.

Monitoring Memory Usage for Each Session

Use the following **onstat -g** arguments to obtain memory information for each session.

Argument	Description
ses	Displays one-line summaries of all active sessions
ses *session id*	Displays session information by *session id*
sql *session id*	Displays SQL information by session If you omit *session id*, this argument displays summaries of all sessions.
stm *session id*	Displays amount of memory used by each prepared SQL statement in a session If you omit *session id*, this argument displays information for all prepared statements.

For examples and discussions of session-monitoring command-line utilities, see "Monitoring Memory Usage for Each Session" on page 13-46 and "Monitoring Sessions and Threads" on page 13-52.

Using SET EXPLAIN

Use the SET EXPLAIN statement or the EXPLAIN directive to display the query plan that the optimizer creates for an individual query. For more information, refer to "Displaying the Query Plan" on page 13-6.

Chapter

3

Effect of Configuration on CPU Utilization

In This Chapter 3-3

UNIX Configuration Parameters That Affect CPU Utilization 3-3
- UNIX Semaphore Parameters 3-4
- UNIX File-Descriptor Parameters 3-6
- UNIX Memory Configuration Parameters 3-6

Windows Configuration Parameters That Affect CPU Utilization . . . 3-7

Configuration Parameters and Environment Variables That Affect CPU Utilization 3-8
- VPCLASS and Other CPU-Related Parameters 3-9
  - VPCLASS 3-9
  - Number of CPU Virtual Processors 3-10
  - MULTIPROCESSOR. 3-11
  - SINGLE_CPU_VP 3-11
  - Process Priority Aging 3-12
  - Processor Affinity 3-12
  - Number of AIO Virtual Processors. 3-15
- OPTCOMPIND. 3-16
- MAX_PDQPRIORITY 3-16
- DS_MAX_QUERIES 3-17
- DS_MAX_SCANS 3-18
- NETTYPE 3-19
  - Specifying the Connection Protocol 3-19
  - Specifying Virtual-Processor Classes for Poll Threads 3-20
  - Specifying Number of Connections and Number of Poll Threads 3-20

Network Buffer Pools 3-22
NETTYPE Configuration Parameter 3-24
IFX_NETBUF_PVTPOOL_SIZE Environment Variable 3-24
IFX_NETBUF_SIZE Environment Variable 3-26

Virtual Processors and CPU Utilization 3-27
Adding Virtual Processors 3-27
Monitoring Virtual Processors 3-27
Using Command-Line Utilities 3-27
Using ISA 3-31
Using SMI Tables 3-31

Connections and CPU Utilization 3-32
Multiplexed Connections 3-32
MaxConnect for Multiple Connections 3-34

In This Chapter

This chapter discusses how the combination of operating-system and database server configuration parameters can affect CPU utilization. This chapter discusses the parameters that most directly affect CPU utilization and describes how to set them. When possible, this chapter also describes considerations and recommends parameter settings that might apply to different types of workloads.

Multiple database server instances that run on the same host computer perform poorly when compared with a single database server instance that manages multiple databases. Multiple database server instances cannot balance their loads as effectively as a single database server. Avoid multiple residency for production environments in which performance is critical.

UNIX

UNIX Configuration Parameters That Affect CPU Utilization

Your database server distribution includes a machine notes file that contains recommended values for UNIX configuration parameters. Compare the values in this file with your current operating-system configuration. For information on where to find this machine notes file, refer to "Additional Documentation" on page 14 in the Introduction.

The following UNIX parameters affect CPU utilization:

- Semaphore parameters
- Parameters that set the maximum number of open file descriptors
- Memory configuration parameters

UNIX Semaphore Parameters

Semaphores are kernel resources with a typical size of 1 byte each. Semaphores for the database server are in addition to any that you allocate for other software packages.

Each instance of the database server requires the following semaphore sets:

- One set for each group of up to 100 virtual processors (VPs) that are initialized with the database server
- One set for each additional VP that you might add dynamically while the database server is running
- One set for each group of 100 or fewer user sessions connected through the shared-memory communication interface

Tip: *For best performance, Informix recommends that you allocate enough semaphores for double the number of* ***ipcshm*** *connections that you expect. Informix also recommends that you use the NETTYPE parameter to configure database server poll threads for this doubled number of connections. For a description of poll threads, refer to your "Administrator's Guide." For information on configuring poll threads, refer to* *"NETTYPE" on page 3-19**.*

Because utilities such as **onmode** use shared-memory connections, you must configure a minimum of two semaphore sets for each instance of the database server: one for the initial set of VPs and one for the shared-memory connections that database server utilities use. The SEMMNI operating-system configuration parameter typically specifies the number of semaphore sets to allocate. For information on how to set semaphore-related parameters, refer to the configuration instructions for your operating system.

The SEMMSL operating-system configuration parameter typically specifies the maximum number of semaphores per set. Set this parameter to at least `100`.

Some operating systems require that you configure a maximum total number of semaphores across all sets, which the SEMMNS operating-system configuration parameter typically specifies. Use the following formula to calculate the total number of semaphores that each instance of the database server requires:

```
SEMMNS = init_vps + added_vps + (2 * shmem_users) + concurrent_utils
```

init_vps is the number of VPs that are initialized with the database server. This number includes CPU, PIO, LIO, AIO, SHM, TLI, SOC, and ADM VPs. (For a description of these VPs, see your *Administrator's Guide*.) The minimum value is 15.

added_vps is the number of VPs that you intend to add dynamically.

shmem_users is the number of shared-memory connections that you allow for this instance of the database server.

concurrent_utils is the number of concurrent database server utilities that can connect to this instance. Informix suggests that you allow for a minimum of six utility connections: two for ON-Archive or ON-Bar and four for other utilities such as ON-Monitor (UNIX only), **onstat**, and **oncheck**.

If you use software packages that require semaphores in addition to those that the database server needs, the SEMMNI configuration parameter must include the total number of semaphore sets that the database server and your other software packages require. You must set the SEMMSL configuration parameter to the largest number of semaphores per set that any of your software packages require. For systems that require the SEMMNS configuration parameter, multiply SEMMNI by the value of SEMMSL to calculate an acceptable value.

UNIX File-Descriptor Parameters

Some operating systems require you to specify a limit on the number of file descriptors that a process can have open at any one time. To specify this limit, use an operating-system configuration parameter, typically NOFILE, NOFILES, NFILE, or NFILES. The number of open file descriptors that each instance of the database server needs depends on the number of chunks in your database, the number of VPs that you run, and the number of network connections that your database server instance must support.

Use the following formula to calculate the number of file descriptors that your instance of the database server requires:

```
NFILES = (chunks * NUMAIOVPS) + NUMCPUVPS + net_connections
```

chunks is the number of chunks to be configured.

net_connections is the number of network connections that you specify in either of the following places:

- **sqlhosts** file or registry
- NETTYPE configuration entries

Network connections include all but those specified as the **ipcshm** connection type.

Each open file descriptor is about the same length as an integer within the kernel. Allocating extra file descriptors is an inexpensive way to allow for growth in the number of chunks or connections on your system.

UNIX Memory Configuration Parameters

The configuration of memory in the operating system can affect other resources, including CPU and I/O. Insufficient physical memory for the overall system load can lead to thrashing, as "Memory Utilization" on page 1-18 describes. Insufficient memory for the database server can result in excessive buffer-management activity. For more information on configuring memory, refer to "Configuring UNIX Shared Memory" on page 4-10.

Windows

Windows Configuration Parameters That Affect CPU Utilization

The Dynamic Server distribution includes a release notes file that contains recommended values for Dynamic Server configuration parameters on Windows NT. Compare the values in this file with your current ONCONFIG configuration file settings. For the pathname of the release notes file, refer to "Additional Documentation" on page 14 in the Introduction.

Dynamic Server runs in the background. For best performance, give the same priority to foreground and background applications.

WIN NT

- Windows NT changes the priorities of threads and processes automatically, but you can use the **Performance** option of **System Properties** icon on the **Control Panel** to change the priorities of foreground and background applications. ♦

WIN 2000

- On Windows 2000, to change the priorities of foreground and background applications, go to **Start→Settings→Control Panel,** open the **System** icon, and click on the **Advanced Tab.** Select the **Performance Options** button and select either the **Applications** or **Background Services** radio button. ♦

The configuration of memory in the operating system can impact other resources, including CPU and I/O. Insufficient physical memory for the overall system load can lead to thrashing, as "Memory Utilization" on page 1-18 describes. Insufficient memory for Dynamic Server can result in excessive buffer-management activity. When you set the **Virtual Memory** values in the **System** icon on the **Control Panel**, ensure that you have enough paging space for the total amount of physical memory.

Configuration Parameters and Environment Variables That Affect CPU Utilization

The following parameters in the database server configuration file have a significant impact on CPU utilization:

- VPCLASS (NUMAIOVPS, NUMCPUVPS, NOAGE)
- MULTIPROCESSOR
- SINGLE_CPU_VP
- OPTCOMPIND
- MAX_PDQPRIORITY
- DS_MAX_QUERIES
- DS_MAX_SCANS
- NETTYPE

The following sections describe how these configuration parameters affect performance. For more information about database server configuration parameters, refer to the *Administrator's Reference*.

The following environment variables affect CPU utilization:

- **OPTCOMPIND**
- **PDQPRIORITY**
- **PSORT_NPROCS**

The **OPTCOMPIND** environment variable, when set in the environment of a client application, indicates the preferred way to perform join operations. This variable overrides the value that the OPTCOMPIND configuration parameter sets. For details on how to select a preferred join method, refer to "OPTCOMPIND" on page 3-16.

The **PDQPRIORITY** environment variable, when set in the environment of a client application, places a limit on the percentage of CPU VP utilization, shared memory, and other resources that can be allocated to any query that the client starts.

A client can also use the SET PDQPRIORITY statement in SQL to set a value for PDQ priority. The actual percentage allocated to any query is subject to the factor that the MAX_PDQPRIORITY configuration parameter sets. For more information on how to limit resources that can be allocated to a query, see "MAX_PDQPRIORITY" on page 3-16.

PSORT_NPROCS, when set in the environment of a client application, indicates the number of parallel sort threads that the application can use. The database server imposes an upper limit of 10 sort threads per query for any application. For more information on parallel sorts and **PSORT_NPROCS**, see "Configuring Dbspaces for Temporary Tables and Sort Files" on page 5-12.

For more information about environment variables that affect Informix database servers, refer to the *Informix Guide to SQL: Reference*.

VPCLASS and Other CPU-Related Parameters

Informix recommends that you use the VPCLASS parameter as an alternative to the following parameters: AFF_SPROC, AFF_NPROCS, NOAGE, NUMCPUVPS, and NUMAIOVPS. When you use VPCLASS, you must explicitly remove these other parameters from your ONCONFIG file. For more information on which configuration parameters to remove, refer to your *Administrator's Guide*.

VPCLASS

The VPCLASS configuration parameter allows you to specify a class of virtual processors, the number of virtual processors that the database server should start for a specific class, and the maximum number allowed.

To execute user-defined routines (UDRs), you can define a new class of virtual processors to isolate UDR execution from other transactions that execute on the CPU virtual processors. Typically you write user-defined routines to support user-defined data types. For more information on the purpose of user-defined virtual processors, refer to your *Administrator's Guide* and the *Creating User-Defined Routines and User-Defined Data Types* manual.

If you do not want a user-defined routine to affect the normal processing of user queries in the CPU class, you can use the CREATE FUNCTION statement to assign the routine to a user-defined class of virtual processors. The class name that you specify in the VPCLASS parameter must match the name specified in the CLASS modifier of the CREATE FUNCTION statement. For more information on the CREATE FUNCTION statement, refer to the *Informix Guide to SQL: Syntax*.

Number of CPU Virtual Processors

The **cpu** and **num** options of the VPCLASS parameter specifies the number of CPU VPs that the database server brings up initially. Do not allocate more CPU VPs than there are CPUs available to service them.

Use the following guidelines to set the number of CPU VPs:

- For uniprocessor computers, Informix recommends that you use one CPU VP.

  ```
  VPCLASS cpu,num=1
  ```

- For multiprocessor systems with four or more CPUs that are primarily used as database servers, Informix recommends that you set the VPCLASS **num** option to one less than the total number of processors. For example, if you have four CPUs, use the following specification:

  ```
  VPCLASS cpu,num=3
  ```

  When you use this setting, one processor is available to run the database server utilities or the client application.

- For multiprocessor systems that you do not use primarily to support database servers, you can start with somewhat fewer CPU VPs to allow for other activities on the system and then gradually add more if necessary.

For dual-processor systems, you might improve performance by running with two CPU VPs. To test if performance improves, set NUMCPUVPS to 1 in your ONCONFIG file and then add a CPU VP dynamically at runtime with **onmode -p**.

MULTIPROCESSOR

If you are running multiple CPU VPs, set the MULTIPROCESSOR parameter to 1. When you set MULTIPROCESSOR to 1, the database server performs locking in a manner that is appropriate for a multiprocessor. Otherwise, set this parameter to 0.

The number of CPU VPs is used as a factor in determining the number of scan threads for a query. Queries perform best when the number of scan threads is a multiple (or factor) of the number of CPU VPs. Adding or removing a CPU VP can improve performance for a large query because it produces an equal distribution of scan threads among CPU VPs. For instance, if you have 6 CPU VPs and scan 10 table fragments, you might see a faster response time if you reduce the number of CPU VPs to 5, which divides evenly into 10. You can use **onstat -g ath** to monitor the number of scan threads per CPU VP or use **onstat -g ses** to focus on a particular session.

SINGLE_CPU_VP

If you are running only one CPU VP, set the SINGLE_CPU_VP configuration parameter to 1. Otherwise, set this parameter to 0.

Important: *If you set the SINGLE_CPU_VP parameter to 1, the value of the NUMCPUVPS parameter must also be 1. If the latter is greater than 1, the database server fails to initialize and displays the following error message:*

```
Cannot have 'SINGLE_CPU_VP' non-zero and 'NUMCPUVPS' greater than 1
```

The database server treats user-defined virtual-processor classes (that is, VPs defined with VPCLASS) as if they were CPU VPS. Thus, if you set SINGLE_CPU_VP to nonzero, you cannot create any user-defined classes.

When you set the SINGLE_CPU_VP parameter to 1, you cannot add CPU VPs while the database server is in online mode.

Process Priority Aging

The **noage** option of the VPCLASS parameter allows you to disable process priority aging for database server CPU VPs on operating systems that support this feature. Priority aging is when the operating system lowers the priority of long-running processes as they accumulate processing time. You might want to disable priority aging because it can cause the performance of the database server processes to decline over time.

Your database server distribution includes a machine notes file that contains information on whether your version of the database server supports this feature. For information on where to find this machine notes file, refer to "Additional Documentation" on page 14 in the Introduction.

Specify the **noage** option of VPCLASS if your operating system supports this feature.

Processor Affinity

The **aff** option of the VPCLASS parameter specifies the processors to which you want to bind CPU VPs or AIO VPs. When you assign a CPU VP to a specific CPU, the VP runs only on that CPU; other processes can also run on that CPU.

The database server supports automatic binding of CPU VPs to processors on multiprocessor host computers that support processor affinity. Your database server distribution includes a machine notes file that contains information on whether your version of the database server supports this feature. For information on where to find this machine notes file, refer to "Additional Documentation" on page 14 in the Introduction.

You can use processor affinity for the purposes that the following sections describe.

Distributing Computation Impact

You can use processor affinity to distribute the computation impact of CPU VPs and other processes. On computers that are dedicated to the database server, assigning CPU VPs to all but one of the CPUs achieves maximum CPU utilization. On computers that support both database server and client applications, you can bind applications to certain CPUs through the operating system. By doing so, you effectively reserve the remaining CPUs for use by database server CPU VPs, which you bind to the remaining CPUs with the VPCLASS configuration parameter. Set the **aff** option of the VPCLASS parameter to the numbers of the CPUs on which to bind CPU VPs. For example, the following VPCLASS setting assigns CPU VPs to processors 4 to 7:

```
VPCLASS cpu,num=4,aff=4-7
```

If you specify a larger number of CPU VPs than physical CPUs, the database server starts assigning CPU VPs from the starting CPU again. For example, suppose you specify the following VPCLASS settings:

```
VPCLASS cpu,num=8,aff=4-7
```

The database server makes the following assignments:

- CPUVP number 0 to CPU 4
- CPUVP number 1 to CPU 5
- CPUVP number 2 to CPU 6
- CPUVP number 3 to CPU 7
- CPUVP number 4 to CPU 4
- CPUVP number 5 to CPU 5
- CPUVP number 6 to CPU 6
- CPUVP number 7 to CPU 7

Isolating AIO VPs from CPU VPs

On a system that runs database server and client (or other) applications, you can bind asynchronous I/O (AIO) VPs to the same CPUs to which you bind other application processes through the operating system. In this way, you isolate client applications and database I/O operations from the CPU VPs. This isolation can be especially helpful when client processes are used for data entry or other operations that require waiting for user input. Because AIO VP activity usually comes in quick bursts followed by idle periods waiting for the disk, you can often interweave client and I/O operations without their unduly impacting each other.

Binding a CPU VP to a processor does not prevent other processes from running on that processor. Application (or other) processes that you do not bind to a CPU are free to run on any available processor. On a computer that is dedicated to the database server, you can leave AIO VPs free to run on any processor, which reduces delays on database operations that are waiting for I/O. Increasing the priority of AIO VPs can further improve performance by ensuring that data is processed quickly once it arrives from disk.

Avoiding a Certain CPU

The database server assigns CPU VPs to CPUs serially, starting with the CPU number you specify in this parameter. You might want to avoid assigning CPU VPs to a certain CPU that has a specialized hardware or operating-system function (such as interrupt handling).

To avoid a certain CPU, set the **aff** option of the VPCLASS parameter to a range of values that excludes that CPU number. For example, the following specification avoids CPU number 2 and assigns CPU VPs to CPU numbers 3, 0, and 1:

```
VPCLASS cpu,num=3,aff=3-1
```

Number of AIO Virtual Processors

The **aio** and **num** options of the VPCLASS parameter indicate the number of AIO VPs that the database server brings up initially. If your operating system does not support kernel asynchronous I/O (KAIO), the database server uses AIO VPs to manage all database I/O requests.

The recommended number of AIO VPs depends on how many disks your configuration supports. If KAIO is *not* implemented on your platform, Informix recommends that you allocate one AIO VP for each disk that contains database tables. You can add an additional AIO VP for each chunk that the database server accesses frequently.

The machine notes file for your version of the database server indicates whether the operating system supports KAIO. If KAIO is supported, the machine notes describe how to enable KAIO on your specific operating system. For information on where to find this machine notes file, refer to "Additional Documentation" on page 14 in the Introduction.

If your operating system supports KAIO, the CPU VPs make I/O requests directly to the file instead of the operating-system buffers. In this case, configure only one AIO VP, plus two additional AIO VPs for every buffered file chunk.

The goal in allocating AIO VPs is to allocate enough of them so that the lengths of the I/O request queues are kept short (that is, the queues have as few I/O requests in them as possible). When the I/O request queues remain consistently short, I/O requests are processed as fast as they occur. The **onstat -g ioq** command allows you to monitor the length of the I/O queues for the AIO VPs.

Allocate enough AIO VPs to accommodate the peak number of I/O requests. Generally, allocating a few extra AIO VPs is not detrimental. To start additional AIO VPs while the database server is in online mode, use the **onmode -p** command. You cannot drop AIO VPs in online mode.

OPTCOMPIND

The OPTCOMPIND parameter helps the optimizer choose an appropriate access method for your application. When the optimizer examines join plans, OPTCOMPIND indicates the preferred method for performing the join operation for an ordered pair of tables. If OPTCOMPIND is equal to `0`, the optimizer gives preference to an existing index (nested-loop join) even when a table scan might be faster. If OPTCOMPIND is set to `1` and the isolation level for a given query is set to Repeatable Read, the optimizer uses nested-loop joins. When OPTCOMPIND is equal to `2` (its default value), the optimizer selects a join method based on cost alone even though table scans can temporarily lock an entire table. For more information on OPTCOMPIND and the different join methods, see "Effect of OPTCOMPIND on the Query Plan" on page 10-26.

To set the value for OPTCOMPIND for specific applications or user sessions, set the **OPTCOMPIND** environment variable for those sessions. Values for this environment variable have the same range and semantics as for the configuration parameter.

MAX_PDQPRIORITY

The MAX_PDQPRIORITY parameter limits the percentage of parallel database query (PDQ) resources that a query can use. Use this parameter to limit the impact of large CPU-intensive queries on transaction throughput.

Specify this parameter as an integer that represents a percentage of the following PDQ resources that a query can request:

- Memory
- CPU VPs
- Disk I/O
- Scan threads

When a query requests a percentage of PDQ resources, the database server allocates the MAX_PDQPRIORITY percentage of the amount requested, as the following formula shows:

```
Resources allocated = PDQPRIORITY/100 * MAX_PDQPRIORITY/100
```

For example, if a client uses the SET PDQPRIORITY 80 statement to request 80 percent of PDQ resources, but MAX_PDQPRIORITY is set to 50, the database server allocates only 40 percent of the resources (50 percent of the request) to the client.

For decision support and online transaction processing (OLTP), setting MAX_PDQPRIORITY allows the database server administrator to control the impact that individual decision-support queries have on concurrent OLTP performance. Reduce the value of MAX_PDQPRIORITY when you want to allocate more resources to OLTP processing. Increase the value of MAX_PDQPRIORITY when you want to allocate more resources to decision-support processing.

For more information on how to control the use of PDQ resources, refer to "Allocating Resources for Parallel Database Queries" on page 12-12.

DS_MAX_QUERIES

The DS_MAX_QUERIES parameter specifies a maximum number of decision-support queries that can run at any one time. In other words, DS_MAX_QUERIES controls only queries whose PDQ priority is nonzero. Queries with a low PDQ priority consume proportionally fewer resources, so a larger number of those queries can run simultaneously. You can use the DS_MAX_QUERIES parameter to limit the performance impact of CPU-intensive queries.

The database server uses the value of DS_MAX_QUERIES with DS_TOTAL_MEMORY to calculate quantum units of memory to allocate to a query. For more information on how the database server allocates memory to queries, refer to "DS_TOTAL_MEMORY" on page 4-18.

DS_MAX_SCANS

The DS_MAX_SCANS parameter limits the number of PDQ scan threads that can run concurrently. This parameter prevents the database server from being flooded with scan threads from multiple decision-support queries.

To calculate the number of scan threads allocated to a query, use the following formula:

```
scan_threads  = min (nfrags, (DS_MAX_SCANS * pdqpriority / 100
   * MAX_PDQPRIORITY / 100) )
```

nfrags is the number of fragments in the table with the largest number of fragments.

pdqpriority is the PDQ priority value set by either the **PDQPRIORITY** environment variable or the SQL statement SET PDQPRIORITY.

Reducing the number of scan threads can reduce the time that a large query waits in the ready queue, particularly when many large queries are submitted concurrently. However, if the number of scan threads is less than *nfrags*, the query takes longer once it is underway.

For example, if a query needs to scan 20 fragments in a table, but the *scan_threads* formula lets the query begin when only 10 scan threads are available, each scan thread scans two fragments serially. Query execution takes approximately twice as long as if 20 scan threads were used.

NETTYPE

The NETTYPE parameter configures poll threads for each connection type that your instance of the database server supports. If your database server instance supports connections over more than one interface or protocol, you must specify a separate NETTYPE parameter for each connection type.

You typically include a separate NETTYPE parameter for each connection type that is associated with a dbservername. You list dbservernames in the DBSERVERNAME and DBSERVERALIASES configuration parameters. You associate connection types with dbservernames through entries in the **sqlhosts** file or registry. For information about connection types and the **sqlhosts** file or registry, refer to your *Administrator's Guide*.

Specifying the Connection Protocol

The first NETTYPE entry for a given connection type applies to all dbservernames associated with that type. Subsequent NETTYPE entries for that connection type are ignored. NETTYPE entries are required for connection types that are used for outgoing communication only even if those connection types are not listed in the **sqlhosts** file or registry.

UNIX

The following protocols apply to UNIX platforms:

- IPCSHM
- TLITCP
- IPCSTR
- SOCTCP ♦

WIN NT

The following protocols apply to Windows NT platforms:

- IPCNMP
- SOCTCP ♦

Specifying Virtual-Processor Classes for Poll Threads

Each poll thread configured or added dynamically by a NETTYPE entry runs in a separate VP. A poll thread can run in one of two VP classes: NET and CPU. For best performance, Informix recommends that you use a NETTYPE entry to assign only one poll thread to the CPU VP class and that you assign all additional poll threads to NET VPs. The maximum number of poll threads that you assign to any one connection type must not exceed NUMCPUVPS.

Specifying Number of Connections and Number of Poll Threads

The optimum number of connections per poll thread is approximately 300 for uniprocessor computers and 350 for multiprocessor computers. However, a poll thread can support 1024 or perhaps more connections.

Each NETTYPE entry configures the number of poll threads for a specific connection type, the number of connections per poll thread, and the virtual-processor class in which those poll threads run, using the following comma-separated fields. There can be no white space within or between these fields.

```
NETTYPE connection_type,poll_threads,c_per_t,vp_class
```

connection_type identifies the protocol-interface combination to which the poll threads are assigned. You typically set this field to match the *connection_type* field of a dbservername entry that is in the **sqlhosts** file or registry.

poll_threads is the number of poll threads assigned to the connection type. Set this value to no more than NUMCPUVPS for any connection type.

c_per_t is the number of connections per poll thread. Use the following formula to calculate this number:

```
c_per_t = connections / poll_threads
```

connections is the maximum number of connections that you expect the indicated connection type to support. For shared-memory connections (**ipcshm**), double the number of connections for best performance.

This field is used only for shared memory connections on Windows NT. Other connection methods on Windows NT ignore this value.

vp_class is the class of virtual processor that can run the poll threads. Specify CPU if you have a single poll thread that runs on a CPU VP. For best performance, specify NET if you require more than one poll thread. If you are running Windows NT, specify NET in all cases. The default value for this field depends on the following conditions:

- If the connection type is associated with the dbservername that is listed in the DBSERVERNAME parameter, and no previous NETTYPE parameter specifies CPU explicitly, the default VP class is CPU. If the CPU class is already taken, the default is NET.
- If the connection type is associated with a dbservername that the DBSERVERALIASES parameter specifies, the default VP class is NET.

If *c_per_t* exceeds 350 and the number of poll threads for the current connection type is less than NUMCPUVPS, you can improve performance by specifying the NET CPU class, adding poll threads (do not exceed NUMCPUVPS), and recalculating *c_per_t*. The default value for *c_per_t* is `50`.

Important: *Each* ***ipcshm*** *connection requires a semaphore. Some operating systems require that you configure a maximum number of semaphores that can be requested by all software packages that run on the computer. For best performance, double the number of actual* ***ipcshm*** *connections when you allocate semaphores for shared-memory communications. Refer to* *"UNIX Semaphore Parameters" on page 3-4*.

If your computer is a uniprocessor and your database server instance is configured for only one connection type, you can omit the NETTYPE parameter. The database server uses the information provided in the **sqlhosts** file or registry to establish client/server connections.

If your computer is a uniprocessor and your database server instance is configured for more than one connection type, include a separate NETTYPE entry for each connection type. If the number of connections of any one type significantly exceeds 300, assign two or more poll threads, up to a maximum of NUMCPUVPS, and specify the NET VP class, as the following example shows:

```
NETTYPE ipcshm,1,200,CPU
NETTYPE tlitcp,2,200,NET # supports 400 connections
```

If your computer is a multiprocessor, your database server instance is configured for only one connection type, and the number of connections does not exceed 350, you can use NETTYPE to specify a single poll thread on either the CPU or the NET VP class. If the number of connections exceeds 350, set the VP class to NET, increase the number of poll threads, and recalculate *c_per_t*.

Network Buffer Pools

The sizes of buffers for TCP/IP connections affect memory and CPU utilization. Sizing these buffers to accommodate a typical request can improve CPU utilization by eliminating the need to break up requests into multiple messages. However, you must use this capability with care; the database server dynamically allocates buffers of the indicated sizes for active connections. Unless you carefully size buffers, they can consume large amounts of memory. For details on how to size network buffers, refer to "IFX_NETBUF_SIZE Environment Variable" on page 3-26.

The database server dynamically allocates network buffers from the global memory pool for request messages from clients. After the database server processes client requests, it returns buffers to a common network buffer pool that is shared among sessions that use SOCTCP, IPCSTR, or TLITCP network connections.

This common network buffer pool provides the following advantages:

- Prevents frequent allocations and deallocations from the global memory pool
- Uses fewer CPU resources to allocate and deallocate network buffers to and from the common network buffer pool for each network transfer
- Reduces contention for allocation and deallocation of shared memory

The free network buffer pool can grow during peak activity periods. To prevent large amounts of unused memory from remaining in these network buffer pools when network activity is no longer high, the database server returns free buffers when the number of free buffers reaches specific thresholds.

The database server provides the following features to further reduce the allocation and deallocation of and contention for the free network buffers:

- A private free network buffer pool for each session to prevent frequent allocations and deallocations of network buffers from the common network buffer pool or from the global memory pool in shared memory
- Capability to specify a larger than 4-kilobyte buffer size to receive network packets or messages from clients

As the system administrator, you can control the free buffer thresholds and the size of each buffer with the following methods:

- NETTYPE configuration parameter
- **IFX_NETBUF_PVTPOOL_SIZE** environment variable
- **IFX_NETBUF_SIZE** environment variable and **b** (client buffer size) option in the **sqlhosts** file or registry

NETTYPE Configuration Parameter

The database server implements a threshold of free network buffers to prevent frequent allocations and deallocations of shared memory for the network buffer pool. This threshold enables the database server to correlate the number of free network buffers with the number of connections that you specify in the NETTYPE configuration parameter.

The database server dynamically allocates network buffers for request messages from clients. After the database server processes client requests, it returns buffers to the network free-buffer pool.

If the number of free buffers is greater than the threshold, the database server returns the memory allocated to buffers over the threshold to the global pool.

The database server uses the following formula to calculate the threshold for the free buffers in the network buffer pool:

```
free network buffers threshold =
   100 + (0.7 * number_connections)
```

The value for *number_connections* is the total number of connections that you specified in the third field of the NETTYPE entry for the different type of network connections (SOCTCP, IPCSTR, or TLITCP). This formula does not use the NETTYPE entry for shared memory (IPCSHM).

If you do not specify a value in the third field of the NETTYPE parameter, the database server uses the default value of 50 connections for each NETTYPE entry corresponding to the SOCTCP, TLITCP, and IPCSTR protocols.

IFX_NETBUF_PVTPOOL_SIZE Environment Variable

The database server provides support for private network buffers for each session that uses SOCTCP, IPCSTR, or TLITCP network connections.

For situations in which many connections and sessions are constantly active, these private network buffers have the following advantages:

- Less contention for the common network buffer pool
- Fewer CPU resources to allocate and deallocate network buffers to and from the common network buffer pool for each network transfer

The **IFX_NETBUF_PVTPOOL_SIZE** environment variable specifies the size of the private network buffer pool for each session. The default size is one buffer. For more information on this environment variable, see the *Informix Guide to SQL: Reference*.

Use the **onstat** options in the following table to monitor the network buffer usage.

Option	Output Field	Description
onstat -g ntu	q-pvt	Current number and highest number of buffers that are free in the private pool for this session
onstat -g ntm	q-exceeds	Number of times the free buffer threshold was exceeded

The **onstat -g ntu** option displays the following format for the **q-pvt** output field:

```
current number / highest number
```

If the number of free buffers (value in **q-pvt** field) is consistently 0, you can perform one of the following actions:

- Increase the number of buffers with the environment variable **IFX_NETBUF_PVTPOOL_SIZE**.
- Increase the size of each buffer with the environment variable **IFX_NETBUF_SIZE**.

The **q-exceeds** field indicates the number of times that the threshold for the shared network free-buffer pool was exceeded. When this threshold is exceeded, the database server returns the unused network buffers (over this threshold) to the global memory pool in shared memory. Optimally, this value should be 0 or a low number so that the server is not allocating or deallocating network buffers from the global memory pool.

IFX_NETBUF_SIZE Environment Variable

The **IFX_NETBUF_SIZE** environment variable specifies the size of each network buffer in the common network buffer pool and the private network buffer pool. The default buffer size is 4 kilobytes.

The **IFX_NETBUF_SIZE** environment variable allows the database server to receive messages longer than 4 kilobytes in one system call. The larger buffer size reduces the amount of overhead required to receive each packet.

Increase the value of **IFX_NETBUF_SIZE** if you know that clients send greater than 4-kilobyte packets. Clients send large packets during any of the following situations:

- Loading a table
- Inserting rows greater than 4 kilobytes
- Sending simple large objects

The **b** option for **sqlhosts** allows the client to send and receive greater than 4 kilobytes. The value for the **sqlhosts** option should typically match the value for **IFX_NETBUF_SIZE**. For more information on **sqlhosts** options, refer to your *Administrator's Guide*.

You can use the following **onstat** command to see the network buffer size:

```
onstat -g afr global | grep net
```

The **size** field in the output shows the network buffer size in bytes.

Virtual Processors and CPU Utilization

While the database server is online, it allows you to start and stop VPs that belong to certain classes. You can use **onmode -p** or Informix Server Administrator (ISA), or ON-Monitor to start additional VPs for the following classes while the database server is online: CPU, AIO, PIO, LIO, SHM, TLI, and SOC. You can drop VPs of the CPU class only while the database server is online.

Adding Virtual Processors

Whenever you add a network VP (SOC or TLI), you also add a poll thread. Every poll thread runs in a separate VP, which can be either a CPU VP or a network VP of the appropriate network type. Adding more VPs can increase the load on CPU resources, so if the NETTYPE value indicates that an available CPU VP can handle the poll thread, the database server assigns the poll thread to that CPU VP. If all the CPU VPs have poll threads assigned to them, the database server adds a second network VP to handle the poll thread.

Monitoring Virtual Processors

Monitor the virtual processors to determine if the number of virtual processors configured for the database server is optimal for the current level of activity.

Using Command-Line Utilities

You can use the following command-line utilities to monitor virtual processors.

onstat -g glo

Use the **onstat -g glo** option to display information about each virtual processor that is currently running as well as cumulative statistics for each virtual-processor class.

The **vps** field in the output shows the number of virtual processors currently active for that class. In the sample output for **onstat -g glo** in Figure 3-1, the **vps** field shows that 3 CPU VPs are currently active.

Use the **onstat -g rea** option, as "onstat -g rea" on page 3-29 describes, to determine if you need to increase the number of virtual processors.

```
MT global info:
sessions threads  vps      lngspins
1        15       8        0

Virtual processor summary:
class    vps      usercpu          syscpu           total
 cpu       3       479.77          190.42          670.18
 aio       1         0.83            0.23            1.07
 pio       1         0.42            0.10            0.52
 lio       1         0.27            0.22            0.48
 soc       0         0.00            0.00            0.00
 tli       0         0.00            0.00            0.00
 shm       0         0.00            0.00            0.00
 adm       1         0.10            0.45            0.55
 opt       0         0.00            0.00            0.00
 msc       1         0.28            0.52            0.80
 adt       0         0.00            0.00            0.00
 total     8       481.67          191.93          673.60

Individual virtual processors:
   vp        pid          class    usercpu          syscpu           total
    1        1776          cpu      165.18           40.50          205.68
    2        1777          adm        0.10            0.45            0.55
    3        1778          cpu      157.83           98.68          256.52
    4        1779          cpu      156.75           51.23          207.98
    5        1780          lio        0.27            0.22            0.48
    6        1781          pio        0.42            0.10            0.52
    7        1782          aio        0.83            0.23            1.07
    8        1783          msc        0.28            0.52            0.80
                           tot      481.67          191.93          673.60
```

Figure 3-1
onstat -g glo Output

onstat -g rea

Use the **onstat -g rea** option to monitor the number of threads in the ready queue. The **status** field in the output shows the value `ready` when the thread is in the ready queue. The **vp-class** output field shows the virtual processor class on which the thread executes. If the number of threads in the ready queue is growing for a class of virtual processors (for example, the CPU class), you might have to add more of those virtual processors to your configuration. Figure 3-2 displays sample **onstat -g rea** output.

```
Ready threads:
tid     tcb      rstcb   prty    status                  vp-class   name

6       536a38   406464  4       ready                     3cpu    main_loop()
28      60cfe8   40a124  4       ready                     1cpu    onmode_mon
33      672a20   409dc4  2       ready                     3cpu    sqlexec
```

Figure 3-2
onstat -g rea Output

onstat -g ioq

Use the **onstat -g ioq** option to determine whether you need to allocate additional AIO virtual processors. The command **onstat -g ioq** displays the length of the I/O queues under the column **len**, as Figure 3-3 shows. You can also see the maximum queue length (since the database server started) in the **maxlen** column. If the length of the I/O queue is growing, I/O requests are accumulating faster than the AIO virtual processors can process them. If the length of the I/O queue continues to show that I/O requests are accumulating, consider adding AIO virtual processors.

```
onstat -g ioq

AIO I/O queues:
q name/id    len maxlen totalops  dskread dskwrite  dskcopy
  adt   0      0      0        0        0        0        0
  msc   0      0      1       12        0        0        0
  aio   0      0      4       89       68        0        0
  pio   0      0      1        1        0        1        0
  lio   0      0      1       17        0       17        0
  kio   0      0      0        0        0        0        0
  gfd   3      0      3      254      242       12        0
  gfd   4      0     17      614      261      353        0


onstat -d
Dbspaces
address  number   flags    fchunk   nchunks  flags    owner    name
a1de1d8  1        1        1        1        N        informix rootdbs
a1df550  2        1        2        1        N        informix space1
 2 active, 32,678 maximum
Chunks
address  chk/dbs offset   size     free     bpages   flags pathname
a1de320  1   1   0        75000    66447             PO-   /ix/root_chunk
a1df698  2   2   0        500      447               PO-   /ix//chunk1
 2 active, 32,678 maximum
```

Figure 3-3
onstat -g ioq and onstat -d Outputs

Each chunk serviced by the AIO virtual processors has one line in the **onstat -g ioq** output, identified by the value `gfd` in the **q name** column. You can correlate the line in **onstat -g ioq** with the actual chunk because the chunks are in the same order as in the **onstat -d** output. For example, in the **onstat -g ioq** output in Figure 3-3, there are two `gfd` queues. The first `gfd` queue holds requests for **root_chunk** because it corresponds to the first chunk shown in the **onstat -d** output in Figure 3-3. Likewise, the second `gfd` queue holds requests for **chunk1** because it corresponds to the second chunk in the **onstat -d** output.

If the database server has a mixture of raw devices and cooked files, the gfd queues correspond only to the cooked files in **onstat -d** output.

Using ISA

To monitor virtual processors with ISA, click on the **VPs** page on the main ISA page. ISA uses information that the **onstat -g glo** command-line option generates. Click **Refresh** to rerun the commands and display fresh information.

Using SMI Tables

You must connect to the **sysmaster** database to query the SMI tables. Query the **sysvpprof** SMI table to obtain information on the virtual processors that are currently running. This table contains the following columns.

Column	Description
vpid	Virtual-processor ID number
class	Virtual-processor class
usercpu	Minutes of user CPU consumed
syscpu	Minutes of system CPU consumed

Connections and CPU Utilization

Some applications have a large number of client/server connections. Opening and closing connections can consume a large amount of system CPU time. The following sections describe ways that you might be able to reduce the system CPU time required to open and close connections.

Multiplexed Connections

Many traditional nonthreaded SQL client applications use multiple database connections to perform work for a single user. Each database connection establishes a separate network connection to the database server.

The multiplexed connection facility for the database server provides the ability for one network connection in the database server to handle multiple database connections from a client application to this database server.

When a nonthreaded client uses a multiplexed connection, the database server still creates the same number of user sessions and user threads as with a nonmultiplexed connection. However, the number of network connections decreases when you use multiplexed connections. Instead, the database server uses a multiplex listener thread to allow the multiple database connections to share the same network connection.

To improve response time for nonthreaded clients, you can use multiplexed connections to execute SQL queries. The amount of performance improvement depends on the following factors:

- The decrease in total number of network connections and the resulting decrease in system CPU time

  The usual cause for a large amount of system CPU time is the processing of system calls for the network connection. Therefore, the maximum decrease in system CPU time is proportional to the decrease in the total number of network connections.

- The ratio of this decrease in system CPU time to the user CPU time

  If the queries are simple and use little user CPU time, you might experience a sizable reduction in response time when you use a multiplexed connection. But if the queries are complex and use a large amount of user CPU time, you might not experience a performance improvement.

  To get an idea of the amounts of system CPU time and user CPU times per virtual processor, use the **onstat -g glo** option.

To use multiplexed connections for a nonthreaded client application, you must take the following steps before you bring up the database server:

1. Add a NETTYPE entry to your ONCONFIG file and specify `SQLMUX` in the **connection_type** field.

   The NETTYPE SQLMUX configuration parameter tells the database server to create the multiplex listener thread. When you specify `SQLMUX` in the **connection_type** field of the NETTYPE configuration parameter, the other NETTYPE fields are ignored.

2. Set the multiplexed option (`m=1`) in the client **sqlhosts** file or registry for the corresponding dbservername entry.

   For more details on the ONCONFIG file NETTYPE entry and the **sqlhosts** entry, refer to your *Administrator's Guide*.

3. On Windows platforms, you must also set the **IFX_SESSION_MUX** environment variable.

Warning: *On Windows, a multithreaded application must not use the multiplexed connection feature. If a multithreaded application enables the multiplexing option in the* ***sqlhosts*** *registry entry and also defines the* ***IFX_SESSION_MUX*** *environment variable, it can produce disastrous results, including crashing and data corruption.*

For more information on restrictions on the use of multiplexed connections, refer to the *Informix ESQL/C Programmer's Manual* and your *Administrator's Guide*.

UNIX

MaxConnect for Multiple Connections

Informix MaxConnect is a networking product for Informix database server environments on UNIX. Use MaxConnect to manage large numbers (from several hundred to tens of thousands) of client / server connections.

MaxConnect is best for OLTP data transfers and not recommended for large multimedia data transfers. MaxConnect provides the following performance advantages for medium to large OLTP configurations:

- Reduces CPU requirements on the database server by reducing the number of physical connections.

  MaxConnect multiplexes connections so that the ratio of client connections to database connections can be 100:1 or higher.
- Improves end-user response time by increasing system scalability to many thousands of connections
- Reduces operating-system overhead by aggregating multiple small packets into one transfer operation

To obtain maximum performance benefit, install MaxConnect on either a dedicated computer to which Informix clients connect or on the client application server. Either of these configurations offloads the CPU requirements of handling a large number of connections from the database server computer.

To monitor MaxConnect, use the **onstat -g imc** command on the database server computer and use the **imcadmin** command on the computer where MaxConnect is located.

For more information about installing, configuring, monitoring, and tuning MaxConnect, see the *Guide to Informix MaxConnect*.

Important: *MaxConnect and the "Guide to Informix MaxConnect" ship separately from Informix Dynamic Server Version 9.3.*

Chapter

4

Effect of Configuration on Memory Utilization

In This Chapter 4-3

Allocating Shared Memory 4-3
- Resident Portion 4-4
- Virtual Portion 4-6
- Message Portion 4-9

Configuring UNIX Shared Memory 4-10

Freeing Shared Memory with onmode -F 4-11

Configuration Parameters That Affect Memory Utilization 4-12
- Setting the Size of the Buffer Pool, Logical-Log Buffer, and Physical-Log Buffer 4-13
- BUFFERS 4-14
- DS_TOTAL_MEMORY 4-18
- LOGBUFF 4-21
- PHYSBUFF 4-21
- LOCKS 4-21
- RESIDENT 4-23
- SHMADD 4-24
- SHMTOTAL 4-24
- SHMVIRTSIZE 4-25
- STACKSIZE 4-26

Parameters That Affect Memory Caches. 4-27
UDR Cache 4-28
Data-Dictionary Cache 4-29
Data-Dictionary Configuration 4-30
Monitoring the Data-Dictionary Cache 4-30
Data-Distribution Cache. 4-31
Data-Distribution Configuration 4-33
Monitoring the Data-Distribution Cache 4-35

SQL Statement Cache 4-37
SQL Statement Cache Configuration 4-39
Monitoring and Tuning the SQL Statement Cache 4-42
Number of SQL Statement Executions 4-42
Monitoring and Tuning the Size of the SQL Statement Cache 4-46
Memory Limit and Size. 4-48
Multiple SQL Statement Cache Pools 4-50
Output Descriptions of onstat Options for SQL Statement Cache. 4-53

Session Memory 4-55

Data-Replication Buffers and Memory Utilization 4-56

Memory-Resident Tables and the Buffer Pool 4-56

Memory Latches 4-58
Monitoring Latches with Command-Line Utilities 4-58
onstat -p 4-58
onstat -s 4-59
Monitoring Latches with ISA 4-59
Monitoring Latches with SMI Tables 4-59

In This Chapter

This chapter discusses the combined effect of operating system and database server configuration parameters on memory utilization. This chapter discusses the parameters that most directly affect memory utilization and explains how to set them. Where possible, this chapter also provides suggested settings or considerations that might apply to different workloads.

Consider the amount of physical memory that is available on your host when you allocate shared memory for the database server. In general, if you increase space for database server shared memory, you can enhance the performance of your database server. You must balance the amount of shared memory dedicated to the database server against the memory requirements for VPs and other processes.

Allocating Shared Memory

You must configure adequate shared-memory resources for the database server in your operating system. Insufficient shared memory can adversely affect performance. When the operating system allocates a block of shared memory, that block is called a *segment*. When the database server attaches all or part of a shared-memory segment, it is called a *portion*.

The database server uses the following shared-memory portions. Each portion makes a separate contribution to the total amount of shared memory that the database server requires:

- Resident portion
- Virtual portion
- Message portion

The resident and message portions are static; you must allocate sufficient memory for them before you bring the database server into online mode. (Typically, you must reboot the operating system to reconfigure shared memory.) The virtual portion of shared memory for the database server grows dynamically, but you must still include an adequate initial amount for this portion in your allocation of operating-system shared memory.

The following sections provide guidelines for estimating the size of each shared-memory portion for the database server so that you can allocate adequate space in the operating system. The amount of space required is the total that all three portions of database server shared memory need, which the SHMTOTAL parameter specifies.

Resident Portion

The resident portion includes areas of shared memory that record the state of the database server, including buffers, locks, log files, and the locations of dbspaces, chunks, and tblspaces. The settings that you use for the following database server configuration parameters help determine the size of this portion:

- BUFFERS
- LOCKS
- LOGBUFF
- PHYSBUFF

In addition to these configuration parameters, which affect the size of the resident portion, the RESIDENT parameter can affect memory utilization. When RESIDENT is set to `1` in the ONCONFIG file of a computer that supports forced residency, the resident portion is never paged out.

The machine notes file for your database server indicates whether your operating system supports forced residency. For information on where to find this machine notes file, refer to "Additional Documentation" on page 14 in the Introduction.

To estimate the size of the resident portion (in kilobytes) when you allocate operating-system shared memory, take the following steps. The result provides an estimate that slightly exceeds the actual memory used for the resident portion.

To estimate the size of the resident portion

1. To estimate the size of the data buffer, use the following formula:

   ```
   buffer_value = (BUFFERS * pagesize) + (BUFFERS * 254)
   ```

   pagesize is the shared-memory page size, as **onstat -b** displays it on the last line in the **buffer size** field.

2. Calculate the values in the following formulas:

   ```
   locks_value = LOCKS * 44
   logbuff_value = LOGBUFF * 1024 * 3
   physbuff_value = PHYSBUFF * 1024 * 2
   ```

3. To calculate the estimated size of the resident portion in kilobytes, use the following formula:

   ```
   rsegsize = (buffer_value + locks_value + logbuff_value
              + physbuff_value + 51,200) / 1024
   ```

Tip: *The LOCKS configuration parameter specifies the initial size of the lock table. If the number of locks that sessions allocate exceeds the value of LOCKS, the database server dynamically increases the size of the lock table. If you expect the lock table to grow dynamically, set SHMTOTAL to 0. When SHMTOTAL is 0, no limit on total memory (resident, virtual, communications, and virtual-extension portions of shared memory) allocation is stipulated.*

For more information about the BUFFERS, LOCKS, LOGBUFF, and PHYSBUFF configuration parameters, see "Configuration Parameters That Affect Memory Utilization" on page 4-12.

Virtual Portion

The virtual portion of shared memory for the database server includes the following components:

- Big buffers, which are used for large read and write I/O operations
- Sort-space pools
- Active thread-control blocks, stacks, and heaps
- User-session data
- Caches for SQL statements, data-dictionary information, and user-defined routines
- A global pool for network-interface message buffers and other information

The SHMVIRTSIZE configuration parameter in the database server configuration file provides the initial size of the virtual portion. As the need for additional space in the virtual portion arises, the database server adds shared memory in increments that the SHMADD configuration parameter specifies, up to a limit on the total shared memory allocated to the database server, which the SHMTOTAL parameter specifies.

The size of the virtual portion depends primarily on the types of applications and queries that you are running. Depending on your application, an initial estimate for the virtual portion might be as low as 100 kilobytes per user or as high as 500 kilobytes per user, plus an additional 4 megabytes if you intend to use data distributions. For guidelines on creating data distributions, refer to the discussion of UPDATE STATISTICS in "Creating Data Distributions" on page 13-15.

The basic algorithm for estimating an initial size of the virtual portion of shared memory is as follows:

```
shmvirtsize = fixed overhead + shared structures +
              (mncs * private structures) +
              other buffers
```

To estimate SHMVIRTSIZE with the preceding formula

1. Use the following formula to estimate the fixed overhead:

```
fixed overhead = global pool +
                 thread pool after booting
```

   Use the **onstat -g mem** command to obtain the pool sizes allocated to sessions. Subtract the value in the **freesize** field from the value in the **totalsize** to obtain the number of bytes allocated per session.

   The *thread pool after booting* variable is partially dependent on the number of virtual processors.

2. Use the following formula to estimate *shared structures*:

```
shared structures = AIO vectors + sort memory +
            dbspace backup buffers +
            data-dictionary cache size +
            size of user-defined routine cache +
            histogram pool +
            STMT_CACHE_SIZE (SQL statement cache) +
            other pools (See onstat display.)
```

   Figure 4-1 lists the location of more information on estimating the size of these shared structures in memory.

Figure 4-1
Finding Information for Shared-Memory Structures

Shared-Memory Structure	Information Location
Sort memory	"Estimating Sort Memory" on page 7-19
Data-dictionary cache	"Data-Dictionary Configuration" on page 4-30
Data-distribution cache (histogram pool)	"Data-Distribution Configuration" on page 4-33
User-defined routine (UDR) cache	"UDR Cache" on page 10-42
SQL statement cache	"Enabling the SQL Statement Cache" on page 13-44 "SQL Statement Cache" on page 4-37
Other pools	To see how much memory is allocated to the different pools, use the **onstat -g mem** command.

3. To estimate the next part of the formula, perform the following steps:

   a. Estimate *mncs* (which is the maximum number of concurrent sessions) with the following formula:

   ```
   mncs = number of poll threads *
      number connections per poll thread
   ```

   The value for number of poll threads is the value that you specify in the second field of the NETTYPE configuration parameter.

   The value for *number of connections per poll thread* is the value that you specify in the third field of the NETTYPE configuration parameter.

   You can also obtain an estimate of the maximum number of concurrent sessions when you execute the **onstat -u** command during peak processing. The last line of the **onstat -u** output contains the maximum number of concurrent user threads.

   b. Estimate the private structures with the following formula:

   ```
   private structures = stack + heap +
                  session control-block structures
   ```

stack	is generally 32 kilobytes but dependent on recursion in user-defined routines. You can obtain the stack size for each thread with the **onstat -g sts** option.
heap	is about 15 kilobytes. You can obtain the heapsize for an SQL statement when you use the **onstat -g stm** option.
session control-block structures	is the amount of memory used per session. The **onstat -g ses** option displays the amount of memory, in bytes, in the **total memory** column listed for each session id.

   For more information on the **onstat -g stm** option, refer to "Session Memory" on page 4-55.

   c. Multiply the results of steps 3a and 3b to obtain the following part of the formula:

   ```
   mncs * private structures
   ```

4. Estimate *other buffers* to account for private buffers allocated for features such as lightweight I/O operations for smart large objects (about 180 kilobytes per user).
5. Add the results of steps 1 through 4 to obtain an estimate for SHMVIRTSIZE.

Tip: *When the database server is running with a stable workload, you can use* ***onstat -g seg*** *to obtain a precise value for the actual size of the virtual portion. You can then use the value for shared memory that this command reports to reconfigure SHMVIRTSIZE.*

Message Portion

The message portion contains the message buffers that the shared-memory communication interface uses. The amount of space required for these buffers depends on the number of user connections that you allow using a given networking interface. If a particular interface is not used, you do not need to include space for it when you allocate shared memory in the operating system. You can use the following formula to estimate the size of the message portion in kilobytes:

```
msegsize = (10,531 * ipcshm_conn + 50,000)/1024
```

ipcshm_conn is the number of connections that can be made using the shared-memory interface, as determined by the NETTYPE parameter for the **ipcshm** protocol.

UNIX

Configuring UNIX Shared Memory

Perform the following steps to configure the shared-memory segments that your database server configuration needs. For information on how to set parameters related to shared memory, refer to the configuration instructions for your operating system.

To configure shared-memory segments for the database server

1. If your operating system does not have a size limit for shared-memory segments, take the following actions:
   a. Set the operating-system configuration parameter for maximum segment size, typically SHMMAX or SHMSIZE, to the total size that your database server configuration requires. This size includes the amount of memory that is required to initialize your database server instance and the amount of shared memory that you allocate for dynamic growth of the virtual portion.
   b. Set the operating-system configuration parameter for the maximum number of segments, typically SHMMNI, to at least 1 per instance of the database server.
2. If your operating system has a segment-size limit, take the following actions:
   a. Set the operating-system configuration parameter for the maximum segment size, typically SHMMAX or SHMSIZE, to the largest value that your system allows.
   b. Use the following formula to calculate the number of segments for your instance of the database server. If there is a remainder, round up to the nearest integer.

      ```
      SHMMNI = total_shmem_size / SHMMAX
      ```

      total_shmem_size is the total amount of shared memory that you allocate for the database server use.

3. Set the operating-system configuration parameter for the maximum number of segments, typically SHMMNI, to a value that yields the total amount of shared memory for the database server when multiplied by SHMMAX or SHMSIZE. If your computer is dedicated to a single instance of the database server, that total can be up to 90 percent of the size of virtual memory (physical memory plus swap space).
4. If your operating system uses the SHMSEG configuration parameter to indicate the maximum number of shared-memory segments that a process can attach, set this parameter to a value that is equal to or greater than the largest number of segments that you allocate for any instance of the database server.

For additional tips on configuring shared memory in the operating system, refer to the machine notes file for UNIX or the release notes file for Windows NT. For the pathname of each file, refer to "Additional Documentation" on page 14 in the Introduction.

Freeing Shared Memory with onmode -F

The database server does not automatically free the shared-memory segments that it adds during its operations. Once memory has been allocated to the database server virtual portion, the memory remains unavailable for use by other processes running on the host computer. When the database server runs a large decision-support query, it might acquire a large amount of shared memory. After the query completes, the database server no longer requires that shared memory. However, the shared memory that the database server allocated to service the query remains assigned to the virtual portion even though it is no longer needed.

The **onmode -F** command locates and returns unused 8-kilobyte blocks of shared memory that the database server still holds. Although this command runs only briefly (one or two seconds), **onmode -F** dramatically inhibits user activity while it runs. Systems with multiple CPUs and CPU VPs typically experience less degradation while this utility runs.

Informix recommends that you run **onmode -F** during slack periods with an operating-system scheduling facility (such as **cron** on UNIX). In addition, consider running this utility after you perform any task that substantially increases the size of database server shared memory, such as large decision-support queries, index builds, sorts, or backup operations. For additional information on the **onmode** utility, refer to the *Administrator's Reference*.

Configuration Parameters That Affect Memory Utilization

The following parameters in the database server configuration file significantly affect memory utilization:

- BUFFERS
- DS_TOTAL_MEMORY
- LOCKS
- LOGBUFF
- MAX_RES_BUFFPCT
- PHYSBUFF
- RESIDENT
- SHMADD
- SHMBASE
- SHMTOTAL
- SHMVIRTSIZE
- STACKSIZE
- Memory cache parameters (see "Parameters That Affect Memory Caches" on page 4-27)
- Network buffer size (see "Network Buffer Pools" on page 3-22)

The SHMBASE parameter indicates the starting address for database server shared memory. When set according to the instructions in the machine notes file or release notes file, this parameter has no appreciable effect on performance. For the pathname of each file, refer to "Additional Documentation" on page 14 in the Introduction.

The following sections describe the performance effects and considerations associated with these parameters. For more information about database server configuration parameters, refer to the *Administrator's Reference*.

Setting the Size of the Buffer Pool, Logical-Log Buffer, and Physical-Log Buffer

The values that you specify for the BUFFERS, DS_TOTAL_MEMORY, LOGBUFF, and PHYSBUFF parameters depend on the type of applications that you are using (OLTP or DSS) and the page size. Figure 4-2 on page 4-13 lists recommended settings for these parameters.

For information on estimating the size of the resident portion of shared memory, see "Resident Portion" on page 4-4. This calculation includes figuring the size of the buffer pool, logical-log buffer, physical-log buffer, and lock table.

Figure 4-2
Guidelines for OLTP and DSS Applications

Parameter	OLTP Applications	DSS Applications
BUFFERS	Set to 20 to 25 percent of the number of megabytes in physical memory. If the levels of paging activity rises, reduce the value of BUFFERS.	Set to a small buffer value and increase the DS_TOTAL_MEMORY value for light scans, queries, and sorts. For operations such as index builds that read data through the buffer pool, configure a larger number of buffers.
DS_TOTAL_MEMORY	Set to 20 to 50 percent of the value of SHMTOTAL, in kilobytes.	Set to 50 to 90 percent of SHMTOTAL.

(1 of 2)

Parameter	OLTP Applications	DSS Applications
LOGBUFF	If you are using unbuffered or ANSI logging, use the **pages/io** value in the logical-log section of the **onstat -l** output for the LOGBUFF value. If you are using buffered logging, keep the **pages/io** value low. The recommended LOGBUFF value is 16 to 32 kilobytes or 64 kilobytes for heavy workloads.	Because database or table logging is usually turned off for DSS applications, set LOGBUFF to 32 kilobytes.
PHYSBUFF	If applications are using physical logging, check the **pages/io** value in the physical-log section of the **onstat -l** output to make sure the I/O activity is not too high. Set PHYSBUFF to a value that is divisible by the page size. The recommended PHYSBUFF value is 16 pages.	Because most DSS applications do not physically log, set PHYSBUFF to 32 kilobytes.

(2 of 2)

BUFFERS

BUFFERS specifies the number of data buffers available to the database server. These buffers reside in the resident portion and are used to cache database data pages in memory.

This parameter has a significant effect on database I/O and transaction throughput. The more buffers that are available, the more likely it is that a needed data page might already reside in memory as the result of a previous request. However, allocating too many buffers can affect the memory-management system and lead to excess paging activity.

The database server uses the following formula to calculate the amount of memory to allocate for this data buffer pool:

```
bufferpoolsize = BUFFERS * page_size
```

page_size is the size of a page in memory for your operating system that **onstat -b** displays on the last line in the **buffer size** field, as the following sample output shows.

```
onstat -b
...

Buffers
address  userthread flgs pagenum  memaddr  nslots pgflgs xflgs owner
waitlist
2 modified, 0 resident, 200 total, 256 hash buckets, 4096 buffer size
```

WIN NT

On Windows NT, the page size is always 4 kilobytes (4096). ♦

Informix suggests that you set BUFFERS between 20 and 25 percent of the number of megabytes in physical memory. For example, if your system has a page size of 2 kilobytes and 100 megabytes of physical memory, you can set BUFFERS to between 10,000 and 12,500, which allocates between 20 megabytes and 25 megabytes of memory.

64-Bit Addressing and BUFFERS

To take advantage of the very large memory available on 64-bit addressing machines, increase the number of buffers in the buffer pool. This larger buffer pool increases the likelihood that a needed data page might already reside in memory.

Smart Large Objects and BUFFERS

By default, the database server reads smart large objects into the buffers in the resident portion of shared memory (also known as the buffer pool).

Depending upon your situation, you can take one of the following actions to achieve better performance for applications that use smart large objects:

- Use the buffer pool by default and increase the value of BUFFERS.

  If your applications frequently access smart large objects that are 2 kilobytes or 4 kilobytes in size, use the buffer pool to keep them in memory longer.

  Use the following formula to increase the value of BUFFERS:

  ```
  Additional_BUFFERS = numcur_open_lo *
                       (lo_userdata / pagesize)
  ```

numcur_open_lo	is the number of concurrently opened smart large objects that you can obtain from the **onstat -g smb fdd** option.
lo_userdata	is the number of bytes of smart-large-object data that you want to buffer.
pagesize	is the page size in bytes for the database server.

  As a general rule, try to have enough buffers to hold two smart-large-object pages for each concurrently open smart large object. (The additional page is available for read-ahead purposes).

- Use lightweight I/O buffers in the virtual portion of shared memory.

  Use lightweight I/O buffers only when you read or write smart large objects in operations greater than 8000 bytes and seldom access them. That is, if the read or write function calls read large amounts of data in a single-function invocation, use lightweight I/O buffers.

  When you use lightweight I/O buffers, you can prevent the flood of smart large objects into the buffer pool and leave more buffers available for other data pages that multiple users frequently access. For more information, refer to "Lightweight I/O for Smart Large Objects" on page 5-32.

Monitoring BUFFERS

You can monitor buffers and buffer-pool activity using the following options of **onstat**:

- **The -b** and **-B** options display general buffer information.
- **The -R** option displays LRU queue statistics.
- **The -X** option displays information about the database server I/O threads that are waiting for buffers.
- The **-p** option displays the number of times the database server attempts to exceed the maximum number of shared buffers specified by the BUFFERS parameter.

You can also use **onstat -p** to monitor the read-cache rate of the buffer pool. This rate represents the percentage of database pages that are already present in a shared-memory buffer when a query requests a page. (If a page is not already present, the database server must copy it into memory from disk.) If the database server finds the page in the buffer pool, it spends less time on disk I/O. Therefore, you want a high read-cache rate for good performance. For OLTP applications where many users read small sets of data, the goal is to achieve a read cache rate of 95 percent or better.

If the read-cache rate is low, you can repeatedly increase BUFFERS and restart the database server. As you increase the value of BUFFERS, you reach a point at which increasing the value no longer produces significant gains in the read-cache rate, or you reach the upper limit of your operating-system shared-memory allocation.

Use the memory-management monitor utility in your operating system (such as **vmstat** or **sar** on UNIX) to note the level of page scans and paging-out activity. If these levels rise suddenly or rise to unacceptable levels during peak database activity, reduce the value of BUFFERS.

DS_TOTAL_MEMORY

The DS_TOTAL_MEMORY parameter places a ceiling on the amount of shared memory that a query can obtain. You can use this parameter to limit the performance impact of large, memory-intensive queries. The higher you set this parameter, the more memory a large query can use, and the less memory is available for processing other queries and transactions.

For OLTP applications, set DS_TOTAL_MEMORY to between 20 and 50 percent of the value of SHMTOTAL, in kilobytes. For applications that involve large decision-support (DSS) queries, increase the value of DS_TOTAL_MEMORY to between 50 and 80 percent of SHMTOTAL. If you use your database server instance exclusively for DSS queries, set this parameter to 90 percent of SHMTOTAL.

A *quantum unit* is the minimum increment of memory allocated to a query. The Memory Grant Manager (MGM) allocates memory to queries in quantum units. The database server uses the value of DS_MAX_QUERIES with the value of DS_TOTAL_MEMORY to calculate a quantum of memory, according to the following formula:

```
quantum = DS_TOTAL_MEMORY / DS_MAX_QUERIES
```

To allow for more simultaneous queries with smaller quanta each, Informix suggests that you increase DS_MAX_QUERIES. For more information on DS_MAX_QUERIES, refer to "DS_MAX_QUERIES" on page 3-17. For more information on the MGM, refer to "Memory Grant Manager" on page 12-10.

Algorithm for Determining DS_TOTAL_MEMORY

The database server derives a value for DS_TOTAL_MEMORY if you do not set DS_TOTAL_MEMORY or if you set it to an inappropriate value. Whenever the database server changes the value that you assigned to DS_TOTAL_MEMORY, it sends the following message to your console:

```
DS_TOTAL_MEMORY recalculated and changed from old_value Kb
           to new_value Kb
```

The variable *old_value* represents the value that you assigned to DS_TOTAL_MEMORY in your configuration file. The variable *new_value* represents the value that the database server derived.

When you receive the preceding message, you can use the algorithm to investigate what values the database server considers inappropriate. You can then take corrective action based on your investigation.

The following sections document the algorithm that the database server uses to derive the new value for DS_TOTAL_MEMORY.

Deriving a Minimum for Decision-Support Memory

In the first part of the algorithm, the database server establishes a minimum for decision-support memory. When you assign a value to the configuration parameter DS_MAX_QUERIES, the database server sets the minimum amount of decision-support memory according to the following formula:

```
min_ds_total_memory = DS_MAX_QUERIES * 128 kilobytes
```

When you do not assign a value to DS_MAX_QUERIES, the database server uses the following formula instead, which is based on the value of VPCLASS cpu or NUMCPUVPS:

```
min_ds_total_memory = NUMCPUVPS * 2 * 128 kilobytes
```

Deriving a Working Value for Decision-Support Memory

In the second part of the algorithm, the database server establishes a working value for the amount of decision-support memory. The database server verifies this amount in the third and final part of the algorithm.

When DS_TOTAL_MEMORY Is Set

The database server first checks whether SHMTOTAL is set. When SHMTOTAL is set, the database server uses the following formula to calculate the amount of decision-support memory:

```
IF DS_TOTAL_MEMORY <= SHMTOTAL - nondecision_support_memory THEN
   decision_support_memory = DS_TOTAL_MEMORY
ELSE
    decision_support_memory = SHMTOTAL -
                  nondecision_support_memory
```

This algorithm effectively prevents you from setting DS_TOTAL_MEMORY to values that the database server cannot possibly allocate to decision-support memory.

When SHMTOTAL is not set, the database server sets decision-support memory equal to the value that you specified in DS_TOTAL_MEMORY.

When DS_TOTAL_MEMORY Is Not Set

When you do not set DS_TOTAL_MEMORY, the database server proceeds as follows. First, the database server checks whether you have set SHMTOTAL. When SHMTOTAL is set, the database server uses the following formula to calculate the amount of decision-support memory:

```
decision_support_memory = SHMTOTAL -
                  nondecision_support_memory
```

When the database server finds that you did not set SHMTOTAL, it sets decision-support memory as in the following example:

```
decision_support_memory = min_ds_total_memory
```

For a description of the variable min_ds_total_memory, refer to "Deriving a Minimum for Decision-Support Memory" on page 4-19.

Checking Derived Value for Decision-Support Memory

The final part of the algorithm verifies that the amount of shared memory is greater than *min_ds_total_memory* and less than the maximum possible memory space for your computer. When the database server finds that the derived value for decision-support memory is less than the value of the *min_ds_total_memory* variable, it sets decision-support memory equal to the value of *min_ds_total_memory*.

When the database server finds that the derived value for decision-support memory is greater than the maximum possible memory space for your computer, it sets decision-support memory equal to the maximum possible memory space.

LOGBUFF

The LOGBUFF parameter determines the amount of shared memory that is reserved for each of the three buffers that hold the logical-log records until they are flushed to the logical-log file on disk. The size of a buffer determines how often it fills and therefore how often it must be flushed to the logical-log file on disk.

If you log smart large objects, increase the size of the logical-log buffers to prevent frequent flushing to the logical-log file on disk.

PHYSBUFF

The PHYSBUFF parameter determines the amount of shared memory that is reserved for each of the two buffers that serve as temporary storage space for data pages that are about to be modified. The size of a buffer determines how often it fills and therefore how often it must be flushed to the physical log on disk. Choose a value for PHYSBUFF that is an even increment of the system page size.

LOCKS

The LOCKS parameter specifies the initial size of the lock table. The lock table holds an entry for each lock that a session uses. If the number of locks that sessions allocate exceeds the value of LOCKS, the database server increases the lock table by doubling its size. Each time that the database server doubles the size of the lock table, it allocates no more than 100,000 locks. The database server can dynamically increase the lock table up to 15 times.

The maximum value for the LOCKS parameter is 8,000,000. The absolute maximum number of locks in the database server is 9,500,000 which is 8,000,000 plus 15 dynamic allocations of 100,000 locks each.

Each lock requires 44 bytes in the resident segment. You must provide for this amount of memory when you configure shared memory.

The default value for the LOCKS configuration parameter is 2000. For more information on when to change this default value, refer to “Configuring and Monitoring the Number of Locks” on page 8-18.

To estimate a different value for the LOCKS configuration parameter, estimate the maximum number of locks that a query needs and multiply this estimate by the number of concurrent users. You can use the guidelines in the following table to estimate the number of locks that a query needs.

Locks per Statement	Isolation Level	Table	Row	Key	TEXT or BYTE Data	CLOB or BLOB Data
SELECT	Dirty Read	0	0	0	0	0
	Committed Read	1	0	0	0	0
	Cursor Stability	1	1	0	0	1 lock for the CLOB or BLOB value or (if byte-range locking is used) 1 lock for each range
	Indexed Repeatable Read	1	Number of rows that satisfy conditions	Number of rows that satisfy conditions	0	1 lock for the CLOB or BLOB value or (if byte-range locking is used) 1 lock for each range
	Sequential Repeatable Read	1	0	0	0	1 lock for the CLOB or BLOB value or (if byte-range locking is used) 1 lock for each range
INSERT	Not applicable	1	1	Number of indexes	Number of pages in TEXT or BYTE data	1 lock for the CLOB or BLOB value
DELETE	Not applicable	1	1	Number of indexes	Number of pages in TEXT or BYTE data	1 lock for the CLOB or BLOB value
UPDATE	Not applicable	1	1	2 per changed key value	Number of pages in old plus new TEXT or BYTE data	1 lock for the CLOB or BLOB value or (if byte-range locking is used) 1 lock for each range

Important: *During the execution of the SQL statement DROP DATABASE, the database server acquires and holds a lock on each table in the database until the entire DROP operation completes. Make sure that the value for LOCKS is large enough to accommodate the largest number of tables in a database.*

RESIDENT

The RESIDENT parameter specifies whether shared-memory residency is enforced for the resident portion of database server shared memory. This parameter works only on computers that support forced residency. The resident portion in the database server contains the buffer pools that are used for database read and write activity. Performance improves when these buffers remain in physical memory. Informix recommends that you set the RESIDENT parameter to 1. If forced residency is not an option on your computer, the database server issues an error message and ignores this parameter.

On machines that support 64-bit addressing, you can have a very large buffer pool and the virtual portion of database server shared memory can also be very large. The virtual portion contains various memory caches that improve performance of multiple queries that access the same tables (see "Parameters That Affect Memory Caches" on page 4-27). To make the virtual portion resident in physical memory in addition to the resident portion, set the RESIDENT parameter to -1.

If your buffer pool is very large, but your physical memory is not very large, you can set RESIDENT to a value greater than 1 to indicate the number of memory segments to stay in physical memory. This specification makes only a subset of the buffer pool resident.

You can turn residency on or off for the resident portion of shared memory in the following ways:

- Use the **onmode** utility to reverse temporarily the state of shared-memory residency while the database server is online.
- Change the RESIDENT parameter to turn shared-memory residency on or off the next time that you initialize database server shared memory.

SHMADD

The SHMADD parameter specifies the size of each increment of shared memory that the database server dynamically adds to the virtual portion. Trade-offs are involved in determining the size of an increment. Adding shared memory consumes CPU cycles. The larger each increment, the fewer increments are required, but less memory is available for other processes. Adding large increments is generally preferred, but when memory is heavily loaded (the scan rate or paging-out rate is high), smaller increments allow better sharing of memory resources among competing programs.

Informix suggests that you set SHMADD according to the size of physical memory, as the following table indicates.

Memory Size	SHMADD Value
256 megabytes or less	8192 kilobytes (the default)
Between 257 and 512 megabytes	16,384 kilobytes
Larger than 512 megabytes	32,768 kilobytes

The size of segments that you add should match those segments allocated in the operating system. For details on configuring shared-memory segments, refer to "Configuring UNIX Shared Memory" on page 4-10. Some operating systems place a lower limit on the size of a shared-memory segment; your setting for SHMADD should be more than this minimum. Use the **onstat -g seg** command to display the number of shared-memory segments that the database server is currently using.

SHMTOTAL

The SHMTOTAL parameter places an absolute upper limit on the amount of shared memory that an instance of the database server can use. If SHMTOTAL is set to `0` or left unassigned, the database server continues to attach additional shared memory as needed until no virtual memory is available on the system.

You can usually leave SHMTOTAL set to `0` except in the following cases:

- You must limit the amount of virtual memory that the database server uses for other applications or other reasons.
- Your operating system runs out of swap space and performs abnormally.

In the latter case, you can set SHMTOTAL to a value that is a few megabytes less than the total swap space that is available on your computer.

SHMVIRTSIZE

The SHMVIRTSIZE parameter specifies the size of the virtual portion of shared memory to allocate when you initialize the database server. The virtual portion of shared memory holds session- and request-specific data as well as other information.

Although the database server adds increments of shared memory to the virtual portion as needed to process large queries or peak loads, allocation of shared memory increases time for transaction processing. Therefore, Informix recommends that you set SHMVIRTSIZE to provide a virtual portion large enough to cover your normal daily operating requirements.

For an initial setting, Informix suggests that you use the larger of the following values:

- `8000`
- *`connections`* `* 350`

The *connections* variable is the number of connections for all network types that are specified in the **sqlhosts** file or registry by one or more NETTYPE parameters. (The database server uses *`connections`* `* 200` by default.)

Once system utilization reaches a stable workload, you can reconfigure a new value for SHMVIRTSIZE. As noted in "Freeing Shared Memory with onmode -F" on page 4-11, you can instruct the database server to release shared-memory segments that are no longer in use after a peak workload or large query.

STACKSIZE

The STACKSIZE parameter indicates the initial stack size for each thread. The database server assigns the amount of space that this parameter indicates to each active thread. This space comes from the virtual portion of database server shared memory.

To reduce the amount of shared memory that the database server adds dynamically, estimate the amount of the stack space required for the average number of threads that your system runs and include that amount in the value that you set for SHMVIRTSIZE. To estimate the amount of stack space that you require, use the following formula:

```
stacktotal = STACKSIZE * avg_no_of_threads
```

avg_no_of_threads is the average number of threads. You can monitor the number of active threads at regular intervals to determine this amount. Use **onstat -g sts** to check the stack use of threads. A general estimate is between 60 and 70 percent of the total number of connections (specified in the NETTYPE parameters in your ONCONFIG file), depending on your workload.

The database server also executes user-defined routines (UDRs) with user threads that use this stack. Programmers who write user-defined routines should take the following measures to avoid stack overflow:

- Do not use large automatic arrays.
- Avoid excessively deep calling sequences.
- Use **mi_call** to manage recursive calls. ♦

DB API

If you cannot avoid stack overflow with these measures, use the STACK modifier of the CREATE FUNCTION statement to increase the stack for a particular routine. For more information on the CREATE FUNCTION statement, refer to the *Informix Guide to SQL: Syntax*.

Parameters That Affect Memory Caches

The database server uses caches to store information in memory instead of performing a disk read to obtain the data or performing some other operation to create the information. These memory caches improve performance for multiple queries that access the same tables.

You can specify configuration parameters to tune the effectiveness of each cache, as the following table shows.

Cache Name	Cache Description	Configuration Parameter	Configuration Parameter Description
Data Dictionary (For details, refer to "Data-Dictionary Cache" on page 4-29)	Stores information about the table definition (such as column names and data types)	DD_HASHSIZE	Number of buckets in data dictionary cache
		DD_HASHMAX	Number of tables that the database server can store in each bucket
Data Distribution (For details, refer to "Data-Distribution Cache" on page 4-31)	Stores distribution statistics for a column	DS_POOLSIZE	Total number of column distribution statistics that the database server can store in the data distribution cache
		DS_HASHSIZE	Number of buckets in data distribution cache

(1 of 2)

Cache Name	Cache Description	Configuration Parameter	Configuration Parameter Description
SQL Statement (For details, refer to "SQL Statement Cache Configuration" on page 4-39)	Stores parsed and optimized SQL statements	STMT_CACHE	Enable the SQL statement cache
		STMT_CACHE_HITS	Number of times SQL statement is executed before caching
		STMT_CACHE_NOLIMIT	Prohibit entries into the SQL statement cache when allocated memory will exceed STMT_CACHE_SIZE
		STMT_CACHE_NUMPOOL	Number of memory pools for the SQL statement cache
		STMT_CACHE_SIZE	Size of the SQL statement cache in kilobytes
UDR (For details, refer to"UDR Cache" on page 10-42	Stores frequently used UDRs (SPL routines and external routines)	PC_POOLSIZE	Total number of user-defined routines and SPL routines in User-Defined Routine cache
		PC_HASHSIZE	Number of buckets in the User-Defined Routine cache

(2 of 2)

UDR Cache

The UDR cache stores frequently used UDRs (SPL routines and external routines). The PC_POOLSIZE configuration parameter determines the size of the UDR cache and most other caches. For more information about the UDR cache and the PC_POOLSIZE, refer to "UDR Cache" on page 10-42.

Data-Dictionary Cache

The first time that the database server accesses a table, it retrieves the information that it needs about the table (such as the column names and data types) from the system catalog tables on disk. Once the database server has accessed the table, it places that information in the data-dictionary cache in memory.

Figure 4-3 shows how the database server uses this cache for multiple users. User 1 accesses the column information for **tabid 120** for the first time. The database server puts the column information in the data-dictionary cache. When user 2, user 3 and user 4 access the same table, the database server does not have to read from disk to access the data-dictionary information for the table. Instead, it reads the dictionary information from the data-dictionary cache in memory.

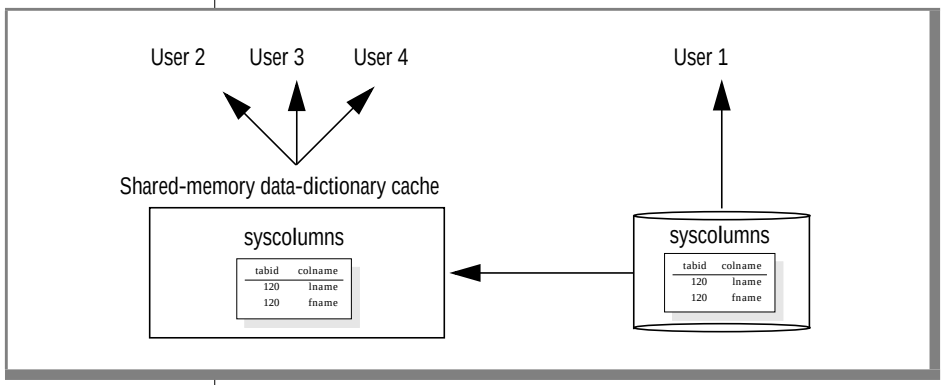


Figure 4-3
Data-Dictionary Cache

The database server still places pages for system catalog tables in the buffer pool, as it does all other data and index pages. However, the data-dictionary cache offers an additional performance advantage, because the data-dictionary information is organized in a more efficient format and organized to allow fast retrieval.

Data-Dictionary Configuration

The database server uses a hashing algorithm to store and locate information within the data-dictionary cache. The DD_HASHSIZE and DD_HASHMAX configuration parameters control the size of the data-dictionary cache. To modify the number of buckets in the data-dictionary cache, use DD_HASHSIZE (must be a prime number). To modify the number of tables that can be stored in one bucket, use DD_HASHMAX.

For medium to large systems, you can start with the following values for these configuration parameters:

- DD_HASHSIZE 503
- DD_HASHMAX 4

With these values, you can potentially store information about 2012 tables in the data-dictionary cache, and each hash bucket can have a maximum of 4 tables.

If the bucket reaches the maximum size, the database server uses a least recently used mechanism to clear entries from the data dictionary.

Monitoring the Data-Dictionary Cache

Use **onstat -g dic** to monitor the data-dictionary cache. If commonly used tables are not listed in the data-dictionary cache, try increasing its size.

Figure 4-4 shows sample output for **onstat -g dic**.

```
Dictionary Cache: Number of lists: 31, Maximum list size: 10

list#  size  refcnt  dirty?  heapptr        table name
--------------------------------------------------------
   9     1      0      no     a210330   dawn@atlanta:informix.sysprocedures

  16     1      0      no     a46a420   dawn@atlanta:informix.orders

Total number of dictionary entries: 2
```

Figure 4-4
onstat -g dic Output

The **onstat -g dic** output has the following fields.

Field	Description
Number of Lists	Number of buckets that DD_HASHSIZE specifies
Maximum List Size	Number of tables allowed in each bucket
List #	Bucket number
Size	Number of tables in the bucket
Ref cnt	Number of times that users have used the data-dictionary information for this table from the cache
Dirty	Designation if the data-dictionary information is no longer valid
Heap ptr	Heap pointer
Table name	Name of the table that the data-dictionary information describes

Data-Distribution Cache

The optimizer uses distribution statistics generated by the UPDATE STATISTICS statement in the MEDIUM or HIGH mode to determine the query plan with the lowest cost. The first time that the optimizer accesses the distribution statistics for a column, the database server retrieves the statistics from the **sysdistrib** system catalog table on disk. Once the database server has accessed the distribution statistics, it places that information in the data-distribution cache in memory.

Figure 4-5 shows how the database server accesses the data-distribution cache for multiple users. When the optimizer accesses the column distribution statistics for User 1 for the first time, the database server puts the distribution statistics in the data-distribution cache. When the optimizer determines the query plan for user 2, user 3 and user 4 who access the same column, the database server does not have to read from disk to access the data-distribution information for the table. Instead, it reads the distribution statistics from the data-distribution cache in memory.

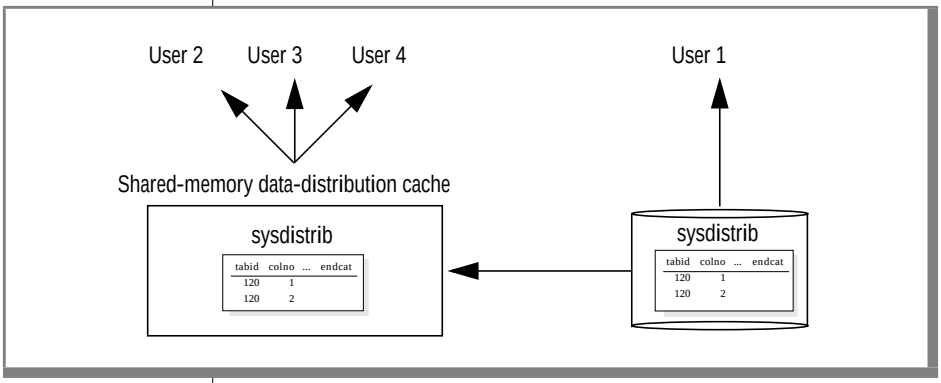


Figure 4-5
Data-Distribution Cache

The database server initially places pages for the **sysdistrib** system catalog table in the buffer pool as it does all other data and index pages. However, the data-distribution cache offers additional performance advantages. It:

- Is organized in a more efficient format
- Is organized to allow fast retrieval
- Bypasses the overhead of the buffer pool management
- Frees more pages in the buffer pool for actual data pages rather than system catalog pages
- Reduces I/O operations to the system catalog table

Data-Distribution Configuration

The database server uses a hashing algorithm to store and locate information within the data-distribution cache. The DS_POOLSIZE controls the size of the data-distribution cache and specifies the total number of column distributions that can be stored in the data-distribution cache. To modify the number of buckets in the data-distribution cache, use the DS_HASHSIZE configuration parameter. The following formula determines the number of column distributions that can be stored in one bucket.

```
Distributions_per_bucket = DS_POOLSIZE / DS_HASHSIZE
```

To modify the number of distributions per bucket, change either the DS_POOLSIZE or DS_HASHSIZE configuration parameter.

For example, with the default values of 127 for DS_POOLSIZE and 31 for DS_HASHSIZE, you can potentially store distributions for about 127 columns in the data-distribution cache. The cache has 31 hash buckets, and each hash bucket can have an average of 4 entries.

The values that you set for DS_HASHSIZE and DS_POOLSIZE, depend on the following factors:

- The number of columns for which you execute UPDATE STATISTICS in HIGH or MEDIUM mode and you expect to be used most often in frequently executed queries.

  If you do not specify columns when you run UPDATE STATISTICS for a table, the database server generates distributions for all columns in the table.

  You can use the values of DD_HASHSIZE and DD_HASHMAX as guidelines for DS_HASHSIZE and DS_POOLSIZE. The DD_HASHSIZE and DD_HASHMAX specify the size for the data-dictionary cache, which stores information and statistics about tables that queries access.

For medium to large systems, you can start with the following values:

- DD_HASHSIZE 503
- DD_HASHMAX 4
- DS_HASHSIZE 503
- DS_POOLSIZE 2000

Monitor these caches to see the actual usage, and you can adjust these parameters accordingly. For monitoring information, refer to "Monitoring the Data-Distribution Cache" on page 4-35.

- The amount of memory available

  The amount of memory required to store distributions for a column depends on the level at which you run UPDATE STATISTICS. Distributions for a single column might require between 1 kilobyte and 2 megabytes, depending on whether you specify medium or high mode or enter a finer resolution percentage when you run UPDATE STATISTICS.

If the size of the data-distribution cache is too small, the following performance problems can occur:

- The database server uses the DS_POOLSIZE value to determine when to remove entries from the data-distribution cache. However, if the optimizer needs the dropped distributions for another query, the database server must reaccess them from the sysdistrib system catalog table on disk. The additional I/O and buffer pool operations to access sysdistrib on disk adds to the total response time of the query.

  The database server tries to maintain the number of entries in data-distribution cache at the DS_POOLSIZE value. If the total number of entries reaches within an internal threshold of DS_POOLSIZE, the database server uses a least recently used mechanism to remove entries from the data-distribution cache. The number of entries in a hash bucket can go past this DS_POOLSIZE value, but the database server eventually reduces the number of entries when memory requirements drop.

- If DS_HASHSIZE is small and DS_POOLSIZE is large, overflow lists can be long and require more search time in the cache.

  Overflow occurs when a hash bucket already contains an entry. When multiple distributions hash to the same bucket, the database server maintains an overflow list to store and retrieve the distributions after the first one.

  If DS_HASHSIZE and DS_POOLSIZE are approximately the same size, the overflow lists might be smaller or even nonexistent, which might waste memory. However, the amount of unused memory is insignificant overall.

Monitoring the Data-Distribution Cache

To monitor the size and use of the data-distribution cache, run onstat -g dsc or use the **ISA Performance -> Cache** menu options. You might want to change the values of DS_HASHSIZE and DS_POOLSIZE if you see the following situations:

- If the data-distribution cache is full most of the time and commonly used columns are not listed in the **distribution name** field, try increasing the values of DS_HASHSIZE and DS_POOLSIZE.
- If the total number of entries is much lower than DS_POOLSIZE, you can reduce the values of DS_HASHSIZE and DS_POOLSIZE.

Figure 4-4 shows sample output for **onstat -g dsc**.

```
onstat -g dsc


Distribution Cache:
    Number of lists             : 31
    DS_POOLSIZE                 : 127

Distribution Cache Entries:

list#id ref_cnt dropped? heap_ptr  distribution name
----------------------------------------------------------------

5     0       0       0   aa8f820  vjp_stores@gilroy:virginia.orders.order_num

12    0       0       0   aa90820  vjp_stores@gilroy:virginia.items.order_num

15    0       0       0   a7e9a38
 vjp_stores@gilroy:virginia.customer.customer_num

19    0       0       0   aa3bc20  vjp_stores@gilroy:virginia.customer.lname

21    0       0       0   aa3cc20  vjp_stores@gilroy:virginia.orders.customer_num

28    0       0       0   aa91820  vjp_stores@gilroy:virginia.customer.company


Total number of distribution entries: 6.
    Number of entries in use    : 0
```

Figure 4-6
onstat -g dsc Output

The **onstat -g dsc** output has the following fields.

Field	Description
Number of Lists	Number of buckets or lists that DS_HASHSIZE specifies
DS_POOLSIZE	Number of column distributions allowed in data-distribution cache
Number of entries	Number of column distributions currently in the data-distribution cache
Number of entries in use	Number of column distributions currently in use
List #	Hash bucket number
Id	Not used
Ref cnt	Number of SQL statements currently referencing the data-distribution information for this column from the cache
Dropped?	Designation if the column distribution has been dropped with the DROP DISTRIBUTIONS keyword on the UPDATE STATISTICS statement.
Heap ptr	Heap pointer
Distribution name	Name of the table and column that the data-distribution information describes

SQL Statement Cache

The SQL statement cache stores parsed and optimized SQL statements so that multiple users who execute the same SQL statement can realize the following performance improvements:

- Reduced response times, because they bypass the parse and optimization steps, as Figure 4-7 shows
- Reduced memory usage, because the database server shares query data structures among users

For more information about the effect of the SQL statement cache on the performance of individual queries, refer to "SQL Statement Cache" on page 13-41.

Figure 4-7 shows how the database server accesses the SQL statement cache for multiple users.

- When the database server executes an SQL statement for User 1 for the first time, the database server checks whether the same exact SQL statement is in the SQL statement cache. If it is not in the cache, the database server parses the statement, determines the optimal query plan, and executes the statement.
- When User 2 executes the same exact SQL statement, the database server finds the statement in the SQL statement cache and does not need to parse and optimize the statement.
- Similarly, if User 3 and User 4 execute the same exact SQL statement, the database server does not have to parse and optimize the statement. Instead, it uses the parse information and query plan in the SQL statement cache in memory.

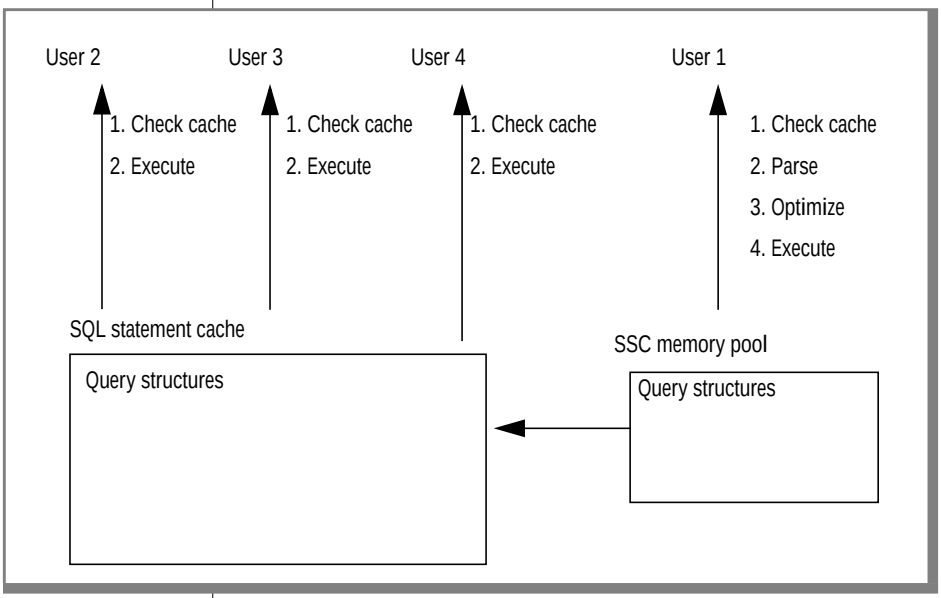


Figure 4-7 *Database Server Actions When Using the SQL Statement Cache*

SQL Statement Cache Configuration

The value of the STMT_CACHE configuration parameter enables or disables the SQL statement cache, as section "Enabling the SQL Statement Cache" on page 13-44 describes.

Figure 4-8 shows how the database server uses the values of the pertinent configuration parameters for the SQL statement cache. Further explanation follows the figure.

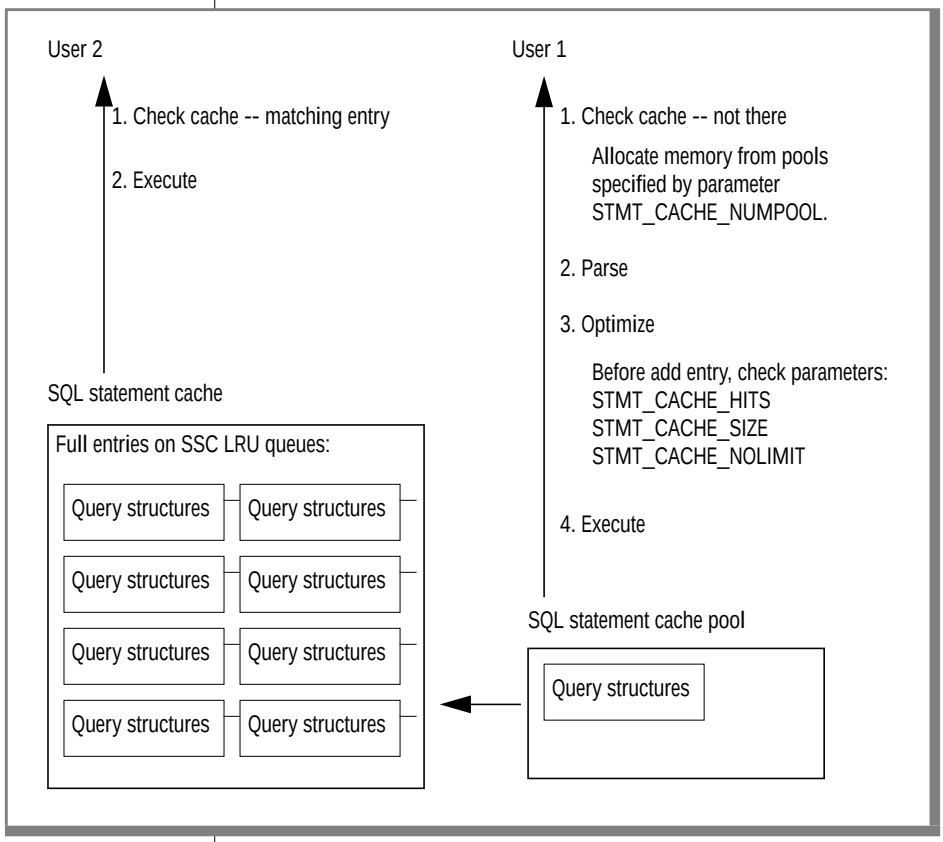


Figure 4-8 *Configuration Parameters That Affect the SQL Statement Cache*

When the database server uses the SQL statement cache for a user, it means the database server takes the following actions:

- Checks the SQL statement cache first for a match of the SQL statement that the user is executing
- If the SQL statement matches an entry, executes the statement using the query memory structures in the SQL statement cache (User 2 in Figure 4-8)
- If the SQL statement does not match an entry, the database server checks if it qualifies for the cache.

  For information on what qualifies an SQL statement for the cache, refer to the SET STATEMENT CACHE statement in the *Informix Guide to SQL: Syntax*.
- If the SQL statement qualifies, inserts an entry into the cache for subsequent executions of the statement.

The following parameters affect whether or not the database server inserts the SQL statement into the cache (User 1 in Figure 4-8 on page 4-39):

- STMT_CACHE_HITS specifies the number of times the statement executes with an entry in the cache (referred to as *hit count*). The database server inserts one of the following entries, depending on the hit count:
  - If the value of STMT_CACHE_HITS is 0, inserts a fully cached entry, which contains the text of the SQL statement plus the query memory structures
  - If the value of STMT_CACHE_HITS is not 0 and the statement does not exist in the cache, inserts a key-only entry that contains the text of the SQL statement. Subsequent executions of the SQL statement increment the hit count.
  - If the value of STMT_CACHE_HITS is equal to the number of hits for a key-only entry, adds the query memory structures to make a fully cached entry.

- STMT_CACHE_SIZE specifies the size of the SQL statement cache, and STMT_CACHE_NOLIMIT specifies whether or not to limit the memory of the cache to the value of STMT_CACHE_SIZE. If you do not specify the STMT_CACHE_SIZE parameter, it defaults to 524288 (512 * 1024) bytes.

  The default value for STMT_CACHE_NOLIMIT is `1`, which means the database server will insert entries into the SQL statement cache even though the total amount of memory might exceed the value of STMT_CACHE_SIZE.

  When STMT_CACHE_NOLIMIT is set to `0`, the database server inserts the SQL statement into the cache if the current size of the cache will not exceed the memory limit.

The following sections provide more details on how the following configuration parameters affect the SQL statement cache and reasons why you might want to change their default values.

- STMT_CACHE_HITS
- STMT_CACHE_SIZE
- STMT_CACHE_NOLIMIT
- STMT_CACHE_NUMPOOL

Monitoring and Tuning the SQL Statement Cache

You can monitor and tune various characteristics of the SQL statement cache. The following table shows the tools you can use to monitor the different characteristics. ISA uses information that the following **onstat** command-line options generate to display information about the SQL statement cache. Click the **Refresh** button to rerun the **onstat** command and display fresh information.

SSC Characteristic to Monitor	Select on ISA	Displays the Output of	Refer to
Number of times statements are read from the cache	**Performance →Cache →Statement Cache**	**onstat -g ssc** **onstat -g ssc all**	"Number of SQL Statement Executions" on page 4-42
Size of the SQL statement cache	**Performance →Cache →Statement Cache**	**onstat -g ssc** **onstat -g ssc all**	"Monitoring and Tuning the Size of the SQL Statement Cache" on page 4-46
Amount of memory used	**Performance →Cache →Statement Cache**	**onstat -g ssc** **onstat -g ssc all**	"Memory Limit and Size" on page 4-48
Usage of the SQL statement cache pools	**Performance →Locks →Latches**	**onstat -g spi**	"Multiple SQL Statement Cache Pools" on page 4-50

Number of SQL Statement Executions

When the SQL statement cache is enabled, the database server inserts a qualified SQL statement and its memory structures immediately in the SQL statement cache by default. If your workload has a disproportionate number of ad hoc queries, use the STMT_CACHE_HITS configuration parameter to specify the number of times an SQL statement is executed before the database server places a fully cached entry in the statement cache.

When the STMT_CACHE_HITS configuration parameter is greater than 0 and the number of times the SQL statement has been executed is less than STMT_CACHE_HITS, the database server inserts key-only entries in the cache. This specification minimizes unshared memory structures from occupying the statement cache, which leaves more memory for SQL statements that applications use often.

Monitor the number of hits on the SQL statement cache to determine if your workload is using this cache effectively. The following sections describe ways to monitor the SQL statement cache hits.

Using onstat -g ssc to Monitor the Number of Hits on the SSC

The **onstat -g ssc** option displays fully cached entries in the SQL statement cache. Figure 4-9 shows sample output for **onstat -g ssc**.

Figure 4-9
onstat -g ssc Output

```
onstat -g ssc

Statement Cache Summary:
#lrus   currsize  maxsize   Poolsize  #hits   nolimit
4       49456     524288    57344     0       1

Statement Cache Entries:

lru hash ref_cnt hits flag heap_ptr      database           user
-----------------------------------------------------------------------
  0  153       0    0   -F  a7e4690      vjp_stores         virginia
  SELECT * FROM customer, orders
    WHERE customer.customer_num = orders.customer_num
    AND order_date > "01/01/97"

  1  259       0    0   -F  aa58c20      vjp_stores         virginia
  SELECT * FROM customer, orders
    WHERE customer.customer_num = orders.customer_num
    AND order_date > "01/01/1997"

  2  232       0    1   DF  aa3d020      vjp_stores         virginia
  SELECT C.customer_num, O.order_num
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
    AND O.order_num = I.order_num

  3  232       1    1   -F  aa8b020      vjp_stores         virginia
  SELECT C.customer_num, O.order_num
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
    AND O.order_num = I.order_num


    Total number of entries: 4.
```

Tip: *The* ***onstat -g ssc*** *option is equivalent to the* ***onstat -g cac stmt*** *option in Version 9.2. You can still issue the* ***onstat -g cac stmt*** *option in this version of the database server, and it displays the same columns as* ***onstat -g ssc****.*

To monitor the number of times that the database server reads the SQL statement within the cache, look at the following output columns:

- In the `Statement Cache Summary` portion of the **onstat -g ssc** output, the **#hits** column is the value of the SQL_STMT_HITS configuration parameter.

  In Figure 4-9 on page 4-43, the **#hits** column in the Statement Cache Summary portion of the output has a value of `0`, which is the default value of the STMT_CACHE_HITS configuration parameter.

Important: *The database server uses entries in the SQL statement cache only if the statements are exactly the same. The first two entries in Figure 4-9 on page 4-43 are not the same because each contains a different literal value in the* ***order_date*** *filter.*

- In the `Statement Cache Entries` portion of the **onstat -g ssc** output, the hits column shows the number of times that the database server executed each individual SQL statement from the cache. In other words, the number of times that the database server uses the memory structures in the cache instead of parsing and optimizing the statement to generate them again.

  The first time it inserts the statement in the cache, the **hits** value is `0`.

  - The first two SQL statements in Figure 4-9 have a **hits** column value of `0`, which indicates that each statement has been inserted into the cache but not yet executed from the cache.
  - The last two SQL statements in Figure 4-9 have a **hits** column value of `1`, which indicates that these statements have been executed once from the cache.

  The **hits** value for individual entries indicates how much sharing of memory structures is done. Higher values in the **hits** column indicates that the SQL statement cache is useful in improving performance and memory usage.

For a complete description of the output fields that **onstat -g ssc** displays, refer to "Output Descriptions of onstat Options for SQL Statement Cache" on page 4-53.

Using onstat -g ssc all

Use the **onstat -g ssc all** option to determine how many nonshared entries exist in the cache. The **onstat -g ssc all** option displays the key-only entries in addition to the fully cached entries in the SQL statement cache.

To determine how many nonshared entries exist in the cache

1. Compare the **onstat -g ssc all** output with the **onstat -g ssc** output.
2. If the difference between these two outputs shows that many nonshared entries exist in the SQL statement cache, increase the value of the STMT_CACHE_HITS configuration parameter to allow more shared statements to reside in the cache and reduce the management overhead of the SQL statement cache.

You can use one of the following methods to change the STMT_CACHE_HITS parameter value:

- Update the ONCONFIG file to specify the STMT_CACHE_HITS configuration parameter. You must restart the database server for the new value to take effect.

  You can use one of the following methods to update the ONCONFIG file:

  - On ISA, navigate to the **Configuration** page and add the STMT_CACHE_HITS there. Then navigate to the **Mode** page and restart the database server.
  - Use a text editor to edit the ONCONFIG file. Then bring down the database server with the **onmode -ky** command and restart with the **oninit** command.

- Use the **onmode -W** command or the ISA **mode** page to increase the STMT_CACHE_HITS configuration parameter dynamically while the database server is running.

  ```
  onmode -W STM_CACHE_HITS 2
  ```

  If you restart the database server, the value reverts the value in the ONCONFIG file. Therefore, if you want the setting to remain for subsequent restarts, modify the ONCONFIG file.

Monitoring and Tuning the Size of the SQL Statement Cache

If the size of the SQL statement cache is too small, the following performance problems can occur:

- Frequently executed SQL statements are not in the cache

  The statements used most often should remain in the SQL statement cache. If the SQL statement cache is not large enough, the database server might not have enough room to keep these statements when other statements come into the cache. For subsequent executions, the database server must reparse, reoptimize, and reinsert the SQL statement into the cache. Try increasing STMT_CACHE_SIZE.

- The database server spends a lot of time cleaning the SQL statement cache

  The database server tries to prevent the SQL statement cache from allocating large amounts of memory by using a threshold (70 percent of the STMT_CACHE_SIZE parameter) to determine when to remove entries from the SQL statement cache. If the new entry causes the size of the SQL statement cache to exceed the threshold, the database server removes least recently used entries (that are not currently in use) before inserting the new entry.

  However, if a subsequent query needs the removed memory structures, the database server must reparse and reoptimize the SQL statement. The additional processing time to regenerate these memory structures adds to the total response time of the query.

You can set the size of the SQL statement cache in memory with the STMT_CACHE_SIZE configuration parameter. The value of the parameter is the size in kilobytes. If STMT_CACHE_SIZE is not set, the default value is 512 kilobytes.

The **onstat -g ssc** output shows the value of STMT_CACHE_SIZE in the **maxsize** column. In Figure 4-9, this **maxsize** column has a value of 524288, which is the default value (512 * 1024 = 524288).

Use the **onstat -g ssc** and **onstat -g ssc all** options to monitor the effectiveness of size of the SQL statement cache. If you do not see cache entries for the SQL statements that applications use most, the SQL statement cache might be too small or too many unshared SQL statement occupy the cache. The following sections describe how to determine these situations.

Changing the Size of the SQL Statement Cache

Look at the values in the following output columns in the "Statement Cache Entries" portion of the **onstat -g ssc all** output.

- The **flags** column shows the current status of an SQL statement in the cache.

  A value of F in the second position indicates that the statement is currently fully cached.

  A value of - in the second position indicates that only the statement text (key-only entry) is in the cache). Entries with this - value in the second position appear in the **onstat -g ssc all** but not in the **onstat -g ssc** output.

- The **hits** column shows the number of times the SQL statement has been executed, excluding the first time it is inserted into the cache.

If you do not see fully cached entries for statements that applications use most and the value in the **hits** column is large for the entries that do occupy the cache, then the SQL statement cache is too small.

You can use one of the following methods to change the STMT_CACHE_SIZE parameter value:

- Update the ONCONFIG file to specify the STMT_CACHE_SIZE configuration parameter. You must restart the database server for the new value to take effect.
- Use the **onmode -W** command to override the STMT_CACHE_SIZE configuration parameter dynamically while the database server is running.

  ```
  onmode -W STMT_CACHE_SIZE 1024
  ```

  If you restart the database server, the value reverts the value in the ONCONFIG file. Therefore, if you want the setting to remain for subsequent restarts, modify the ONCONFIG file.

Too Many Single-use Queries in the SQL Statement Cache

When the database server places many queries that are only used once in the cache, they might replace statements that other applications use often.

Look at the values in the following output columns in the `Statement Cache Entries` portion of the **onstat -g ssc all** output. If you see a lot of entries that have both of the following values, too many unshared SQL statements occupy the cache:

- **flags** column value of `F` in the second position

  A value of `F` in the second position indicates that the statement is currently fully cached.

- **hits** column value of `0` or `1`

  The **hits** column shows the number of times the SQL statement has been executed, excluding the first time it is inserted into the cache.

Increase the value of the STMT_CACHE_HITS configuration parameter to prevent unshared SQL statements from being fully cached. For information on how to change the STMT_CACHE_HITS configuration parameter, refer to "Number of SQL Statement Executions" on page 4-42.

Memory Limit and Size

Although the database server tries to clean the SQL statement cache, sometimes entries cannot be removed because they are currently in use. In this case, the size of the SQL statement cache can exceed the value of STMT_CACHE_SIZE.

The default value of the STMT_CACHE_NOLIMIT configuration parameter is `1`, which means the database server inserts the statement even though the current size of the cache might be greater than the value of the STMT_CACHE_SIZE parameter.

If the value of the STMT_CACHE_NOLIMIT configuration parameter is `0`, the database server does not insert either a fully-qualified or key-only entry into the SQL statement cache if the size will exceed the value of STMT_CACHE_SIZE.

Use the **onstat -g ssc** option to monitor the current size of the SQL statement cache. Look at the values in the following output columns of the **onstat -g ssc** output:

- The **currsize** column shows the number of bytes currently allocated in the SQL statement cache.

  In Figure 4-9 on page 4-43, the **currsize** column has a value of 11264.

- The **maxsize** column shows the value of STMT_CACHE_SIZE.

  In Figure 4-9, the **maxsize** column has a value of 524288, which is the default value (512 * 1024 = 524288).

When the SQL statement cache is full and users are currently executing all statements within it, any new SQL statements that a user executes can cause the SQL statement cache to grow beyond the size that STMT_CACHE_SIZE specifies. Once the database server is no longer using an SQL statement within the SQL statement cache, it frees memory in the SQL statement cache until the size reaches a threshold of STMT_CACHE_SIZE. However, if thousands of concurrent users are executing several ad hoc queries, the SQL statement cache can grow very large before any statements are removed. In such cases, take one of the following actions:

- Set the STMT_CACHE_NOLIMIT parameter to 0 to prevent insertions when the cache size exceeds the value of the STMT_CACHE_SIZE parameter.
- Set the STMT_CACHE_HITS parameter to a value greater than 0 to prevent caching unshared SQL statements.

You can use one of the following methods to change the STMT_CACHE_NOLIMIT parameter value:

- Update the ONCONFIG file to specify the STMT_CACHE_NOLIMIT configuration parameter. You must restart the database server for the new value to take effect.
- Use the **onmode -W** command to override the STMT_CACHE_NOLIMIT configuration parameter dynamically while the database server is running.

  ```
  onmode -W STMT_CACHE_NOLIMIT 0
  ```

  If you restart the database server, the value reverts the value in the ONCONFIG file. Therefore, if you want the setting to remain for subsequent restarts, modify the ONCONFIG file.

Multiple SQL Statement Cache Pools

When the SQL statement cache is enabled, the database server allocates memory from one pool (by default) for the query structures in the following situations:

- When the database server does not find a matching entry in the cache
- When the database server finds a matching key-only entry in the cache and the hit count reaches the value of the STMT_CACHE_HITS configuration parameter

This one pool can become a bottleneck as the number of users increases. The STMT_CACHE_NUMPOOL configuration parameter allows you to configure multiple **sscpools**.

You can monitor the pools in the SQL statement cache to determine the following situations:

- The number of sscpools is sufficient for your workload.
- The size or limit of the SQL statement cache is not causing excessive memory management.

Number of SQL Statement Cache Pools

When the SQL statement cache is enabled, the database server allocates memory from an sscpool for unlinked SQL statements. The default value for the STMT_CACHE_NUMPOOL configuration parameter is 1. As the number of users increases, this one sscpool might become a bottleneck. (The number of longspins on the sscpool indicates whether or not the sscpool is a bottleneck.)

Use the **onstat -g spi** option to monitor the number of longspins on an sscpool. The **onstat -g spi** command displays a list of the resources in the system for which a wait was required before a latch on the resource could be obtained. During the wait, the thread spins (or loops), trying to acquire the resource. The **onstat -g spi** output displays the number of times a wait (**Num Waits** column) was required for the resource and the number of total loops (**Num Loops** column). The **onstat -g spi** output displays only resources that have at least one wait.

Figure 4-10 shows an excerpt of sample output for **onstat -g spi**. Figure 4-10 indicates that no waits occurred for any sscpool (the **Name** column does not list any sscpools).

```
Spin locks with waits:
Num Waits   Num Loops   Avg Loop/Wait    Name
34477       387761      11.25            mtcb sleeping_lock
312         10205       32.71            mtcb vproc_list_lock
```

Figure 4-10
onstat -g spi Output

If you see an excessive number of longspins (**Num Loops** column) on an sscpool, increase the number of sscpools in the STMT_CACHE_NUMPOOL configuration parameter to improve performance.

Size of SQL Statement Cache Pools and Current Cache Size

Use the **onstat -g ssc pool** option to monitor the usage of each SQL statement cache pool. The **onstat -g ssc pool** command displays the size of each pool. The **onstat -g ssc** option displays the cumulative size of the SQL statement cache in the **currsize** column. This current size is the size of memory allocated from the sscpools by the statements that are inserted into the cache. Because not all statements that allocate memory from the sscpools are inserted into the cache, the current cache size could be smaller than the total size of the sscpools. Normally, the total size of all sscpools does not exceed the STMT_CACHE_SIZE value.

Figure 4-11 shows sample output for **onstat -g ssc pool**.

```
onstat -g ssc pool


Pool Summary:
name         class addr     totalsize freesize #allocfrag #freefrag
sscpool0     V     a7e4020  57344      2352     52         7

Blkpool Summary:
name         class addr     size      #blks
```

Figure 4-11
onstat -g ssc pool Output

The `Pool Summary` section of the **onstat -g ssc pool** output lists the following information for each pool in the cache**.**

Column	Description
name	The name of the sscpool
class	The shared-memory segment type in which the pool has been created. For sscpools, this value is always "V" for the virtual portion of shared-memory.
addr	The shared-memory address of the SSC pool structure
totalsize	The total size, in bytes, of this SSC pool
freesize	The number of free bytes in this SSC pool
#allocfrag	The number of contiguous areas of memory in this sscpool that are allocated
#freefrag	The number of contiguous areas of memory that are not used in this SSC pool

The `Blkpool Summary` section of the **onstat -g ssc poo**l output lists the following information for all pools in the cache**.**

Column	Description
name	The name of the sscpool
class	The shared-memory segment type in which the pool has been created. For sscpools, this value is always "V" for the virtual portion of shared-memory.
addr	The shared-memory address of the SSC pool structure
size	The total size, in bytes, of all pools in the SSC
#blks	The number of 8-kilobyte blocks that make up all the SSC pools

Output Descriptions of onstat Options for SQL Statement Cache

The **onstat -g ssc** option lists the following summary information for the SQL statement cache**.**

Column	Description
#lrus	The number of LRU queues. Multiple LRU queues facilitate concurrent lookup and insertion of cache entries.
currsize	The number of bytes currently allocated to entries in the SQL statement cache
maxsize	The number of bytes specified in the STMT_CACHE_SIZE configuration parameter
poolsize	The cumulative number of bytes for all pools in the SQL statement cache. Use the **onstat -g ssc pool** option to monitor individual pool usage.
#hits	Current setting of the STMT_CACHE_HITS configuration parameter, which specifies the number of times that a query is executed before it is inserted into the cache
nolimit	Current setting of STMT_CACHE_NOLIMIT configuration parameter

The **onstat -g ssc** option lists the following information for each fully cached entry in the cache**.** The **onstat -g ssc** all option lists the following information for both the fully cached entries and key-only entries.

Column	Description
lru	The LRU identifier
hash	Hash-bucket identifier
ref_cnt	Number of sessions currently using this statement
hits	Number of times that users read the query from the cache (excluding the first time the statement entered the cache)

(1 of 2)

Column	Description
flags	The flag codes for position 1: D Indicates that the statement has been dropped A statement in the cache can be dropped (not used any more) when one of its dependencies has changed. For example, when you run UPDATE STATISTICS for the table, the optimizer statistics might change, making the query plan for the SQL statement in the cache obsolete. In this case, the database server marks the statement as dropped the next time that it tries to use it. - Indicates that the statement has not been dropped The flag codes for position 2 F Indicates that the cache entry is fully cached and contains the memory structures for the query - Indicates that the statement is not fully cached. A statement is not fully cached when the number of times the statement has been executed is less than the value of the STMT_CACHE_HITS configuration parameter. Entries with this - value in the second position appear in the **onstat -g ssc all** but not in the **onstat -g ssc** output.
heap_ptr	Pointer to the associated heap for the statement
database	Database against which the SQL statement is executed
user	User executing the SQL statement
statement	Statement text as it would be used to test for a match

(2 of 2)

Session Memory

The database server uses the virtual portion of shared memory mainly for user sessions. The majority of the memory that each user session allocates is for SQL statements. The amount of used memory can vary from one statement to another.

Use the following utility options to determine which session and prepared SQL statements have high memory utilization:

- **onstat -g mem**
- **onstat -g stm**

The **onstat -g mem** option displays memory usage of all sessions. You can find the session that is using the most memory by looking at the **totalsize** and **freesize** output columns. Figure 4-13 shows sample output for **onstat -g mem**. This sample output shows the memory utilization of three user sessions with the values 14, 16, 17 in the **names** output column.

```
onstat -g mem

Pool Summary:
name         class addr     totalsize freesize #allocfrag #freefrag
...
14           V     a974020  45056     11960    99         10
16           V     a9ea020  90112     10608    159        5
17           V     a973020  45056     11304    97         13
...
Blkpool Summary:
name         class addr     size      #blks
mt           V     a235688  798720    19
global       V     a232800  0         0
```

Figure 4-12
onstat -g mem Output

To display the memory allocated by each prepared statement, use the **onstat -g stm** option. Figure 4-13 shows sample output for **onstat -g stm**.

```
onstat -g stm


session   25 ------------------------------------------------
 sdblock  heapsz  statement ('*' = Open cursor)
 d36b018    9216  select sum(i) from t where i between -1 and ?
 d378018    6240  *select tabname from systables where tabid=7
 d36b114    8400  <SPL statement>
```

Figure 4-13
onstat -g stm Output

The **heapsz** column in the output in Figure 4-13 shows the amount of memory used by the statement. An asterisk (*) precedes the statement text if a cursor is open on the statement. The output does not show the individual SQL statements in an SPL routine.

To display the memory for only one session, specify the session ID in the **onstat -g stm** option. For an example, refer to "onstat -g mem and onstat -g stm" on page 13-55.

Data-Replication Buffers and Memory Utilization

Data replication requires two instances of the database server, a primary one and a secondary one, running on two computers. If you implement data replication for your database server, the database server holds logical-log records in the data-replication buffer before it sends them to the secondary database server. The data-replication buffer is always the same size as the logical-log buffer.

Memory-Resident Tables and the Buffer Pool

If a database contains small tables that are used by applications for which performance is important, you can instruct the database server to leave data and index pages for these tables in the buffer pool. A table whose pages should remain in memory is known as a *memory-resident* table.

Although the least recently used mechanism tends to keep pages that the database server reads often in memory, it does not improve performance in cases where small tables are not accessed often enough to be kept in memory. For more information on the least recently used mechanism, refer to your *Administrator's Guide*.

To specify which tables or fragments should be memory-resident, use the SET TABLE statement in SQL. You must execute this statement every time that you bring up the database server. For more information about the SET TABLE statement, refer to the *Informix Guide to SQL: Syntax*.

You can also specify that indexes be memory resident with the SET INDEX statement in SQL.

Use caution when you determine which tables, fragments, or indexes should be memory resident. If there are too many memory-resident tables, or if the memory-resident tables are too large, the database server might swap out pages that do not belong to memory-resident tables too often. An improper use of memory-resident tables can decrease performance for applications that use tables that are not set to memory resident.

To monitor the effect of memory-resident tables, you can use the following **onstat** options:

- **onstat -P**
- **onstat -b**
- **onstat -B**
- **onstat -p**
- **onstat -g seg**

To list the number of buffer pages that have data or index pages from memory-resident tables, use **onstat -P**. The following excerpt from **onstat -P** output shows three tables, fragments, or indexes that have a total of four memory-resident pages:

```
partnum  total    btree    data     other    resident
0        99       17       21       61       0
1048578  3        1        1        1        0
1048579  6        4        2        0        0
...
1048828  2        1        0        1        2
1048829  1        0        0        1        1
1048830  1        0        0        1        1
```

Also, the **onstat -b** (or **onstat -B**) command shows the total number of memory resident pages at any one time. The following **onstat -b** output shows that there are four memory-resident pages out of a total of 200 pages in the buffer pool:

```
Buffers
address  userthread flgs pagenum  memaddr  nslots pgflgs xflgs owner
waitlist
0 modified, 4 resident, 200 total, 256 hash buckets, 2048 buffer size
```

As usual, you should monitor the buffer cache rate with **onstat -p** because the cache rate can decrease significantly if you use memory-resident tables improperly.

The **onstat -g seg** command shows an asterisk (*) next the memory segments that are resident.

Memory Latches

The database server uses latches to control access to shared memory structures such as the buffer pool or the memory pools for the SQL statement cache.

You can obtain statistics on latch use and information on specific latches. These statistics provide a measure of the system activity. The statistics include the number of times that threads had to wait to obtain a latch. A large number of latch waits typically results from a high volume of processing activity in which the database server is logging most of the transactions.

Information on specific latches includes a listing of all the latches that are held by a thread and any threads that are waiting for latches. This information allows you to locate any specific resource contentions that exist.

You, as the database administrator, cannot configure or tune the number of latches. However, you can increase the number of memory structures on which the database server places latches to reduce the number of latch waits. For example, you can tune the number of SQL statement cache memory pools or the number of SQL statement cache LRU queues. For more information, refer to "Multiple SQL Statement Cache Pools" on page 4-50.

Warning: *Never kill a database server process that is holding a latch. If you do, the database server immediately initiates an abort.*

Monitoring Latches with Command-Line Utilities

You can use the following command-line utilities to obtain information about latches.

onstat -p

Execute **onstat -p** to obtain the values in the field **lchwaits**. This field stores the number of times that a thread was required to wait for a shared-memory latch.

Figure 4-14 shows an excerpt of sample **onstat -p** output that shows the lchwaits field.

```
...
ixda-RA  idx-RA   da-RA    RA-pgsused lchwaits
5        0        204      148        12
...
```

Figure 4-14
onstat -p Output Showing lchwaits Field

onstat -s

Execute **onstat -s** to obtain general latch information. The output includes the **userthread** column, which lists the address of any user thread that is waiting for a latch. You can compare this address with the user addresses in the **onstat -u** output to obtain the user-process identification number.

Figure 4-15 shows sample **onstat -s** output.

```
...
Latches with lock or userthread set
name     address  lock wait userthread
LRU1     402e90   0    0        6b29d8
bf[34]   4467c0   0    0        6b29d8
...
```

Figure 4-15
onstat -s Output

Monitoring Latches with ISA

To monitor latches and spin locks with ISA, navigate to the **Performance -> Locks** page and click **Latches** or **Spin Locks**. ISA uses information generated by **onstat -s** and **onstat -g spi** to display information. Click **Refresh** to rerun the commands and display fresh information.

Monitoring Latches with SMI Tables

Query the **sysprofile** SMI table to obtain the number of times a thread had to wait for a latch. The **latchwts** column contains the number of times that a thread had to wait for a latch.

Chapter

5

Effect of Configuration on I/O Activity

In This Chapter 5-5

Chunk and Dbspace Configuration 5-5
- Associate Disk Partitions with Chunks. 5-6
- Associate Dbspaces with Chunks 5-6
- Place System Catalog Tables with Database Tables. 5-7

Placement of Critical Data 5-7
- Consider Separate Disks for Critical Data Components 5-8
- Consider Mirroring for Critical Data Components. 5-8
  - Mirroring the Root Dbspace 5-9
  - Mirroring Smart-Large-Object Chunks 5-9
  - Mirroring the Logical Log 5-10
  - Mirroring the Physical Log 5-11

Configuration Parameters That Affect Critical Data 5-11

Configuring Dbspaces for Temporary Tables and Sort Files 5-12
- Creating Temporary Dbspaces 5-14
- DBSPACETEMP Configuration Parameter 5-15
- DBSPACETEMP Environment Variable 5-15
- Estimating Temporary Space 5-16
- PSORT_NPROCS Environment Variable 5-17

Configuring Sbspaces for Temporary Smart Large Objects 5-18
- Creating Temporary Sbspaces 5-19
- SBSPACETEMP Configuration Parameter 5-20

Placement of Simple Large Objects 5-20
Advantage of Blobspaces over Dbspaces 5-21
Blobpage Size Considerations 5-22
Optimizing Blobspace Blobpage Size 5-23
Obtaining Blobspace Storage Statistics 5-24
Determining Blobpage Fullness with oncheck -pB 5-24

Parameters That Affect I/O for Smart Large Objects 5-27
Disk Layout for Sbspaces 5-28
Configuration Parameters That Affect Sbspace I/O 5-28
SBSPACENAME 5-28
BUFFERS 5-29
LOGBUFF 5-29
onspaces Options That Affect Sbspace I/O 5-29
Sbspace Extent Sizes 5-30
Lightweight I/O for Smart Large Objects 5-32
Logging 5-34

How the Optical Subsystem Affects Performance 5-34

Environment Variables and Configuration Parameters for the Optical Subsystem 5-35
STAGEBLOB 5-36
OPCACHEMAX 5-36
INFORMIXOPCACHE 5-37

Table I/O 5-37
Sequential Scans 5-37
Light Scans 5-38
Unavailable Data 5-39

Configuration Parameters That Affect Table I/O 5-40
RA_PAGES and RA_THRESHOLD 5-40
DATASKIP 5-41

Background I/O Activities 5-41
Configuration Parameters That Affect Checkpoints 5-42
CKPINTVL 5-42
LOGSIZE and LOGFILES 5-43
PHYSFILE 5-44
ONDBSPDOWN 5-47
Configuration Parameters That Affect Logging 5-48
LOGBUFF and PHYSBUFF 5-48
LOGFILES 5-49
LOGSIZE 5-49
DYNAMIC_LOGS 5-52
LTXHWM and LTXEHWM 5-54
Configuration Parameters That Affect Page Cleaning 5-55
CLEANERS 5-56
LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY 5-56
Configuration Parameters That Affect Backup and Restore 5-57
ON-Bar Configuration Parameters 5-57
ontape Configuration Parameters 5-58
Configuration Parameters That Affect Rollback and Recovery . . . 5-58
Configuration Parameters That Affect Data Replication and Auditing 5-59
Data Replication 5-59
Auditing 5-60

In This Chapter

Database server configuration affects I/O activity in several ways. The assignment of chunks and dbspaces can create I/O *hot spots*, or disk partitions with a disproportionate amount of I/O activity. Your allocation of critical data, sort areas, and areas for temporary files and index builds can place intermittent loads on various disks. How you configure read-ahead can increase the effectiveness of individual I/O operations. How you configure the background I/O tasks, such as logging and page cleaning, can affect I/O throughput. The following sections discuss each of these topics.

Chunk and Dbspace Configuration

All the data that resides in an Informix database is stored on disk. The Optical Subsystem also uses a magnetic disk to access TEXT or BYTE data that is retrieved from optical media. The speed at which the database server can copy the appropriate data pages to and from disk determines how well your application performs.

Disks are typically the slowest component in the I/O path for a transaction or query that runs entirely on one host computer. Network communication can also introduce delays in client/server applications, but these delays are typically outside the control of the database server administrator. For information on actions that the database server administrator can take to improve network communications, refer to "Network Buffer Pools" on page 3-22 and "Connections and CPU Utilization" on page 3-32.

Disks can become overused or saturated when users request pages too often. Saturation can occur in the following situations:

- You use a disk for multiple purposes, such as for both logging and active database tables.
- Disparate data resides on the same disk.
- Table extents become interleaved.

The various functions that your application requires, as well as the consistency-control functions that the database server performs, determine the optimal disk, chunk, and dbspace layout for your application. The more disks that you make available to the database server, the easier it is to balance I/O across them. For more information on these factors, refer to Chapter 6, "Table Performance Considerations."

This section outlines important issues for the initial configuration of your chunks, dbspaces, and blobspaces. Consider the following issues when you decide how to lay out chunks and dbspaces on disks:

- Placement and mirroring of critical data
- Load balancing
- Reduction of contention
- Ease of backup and restore

Associate Disk Partitions with Chunks

Informix recommends that you assign chunks to entire disk partitions. When a chunk coincides with a disk partition (or device), it is easy to track disk-space use, and you avoid errors caused by miscalculated offsets.

Associate Dbspaces with Chunks

Informix recommends that you associate a single chunk with a dbspace, especially when that dbspace is to be used for a table fragment. For more information on table placement and layout, refer to Chapter 6, "Table Performance Considerations."

Place System Catalog Tables with Database Tables

When a disk that contains the system catalog for a particular database fails, the entire database remains inaccessible until the system catalog is restored. Because of this potential inaccessibility, Informix recommends that you do not cluster the system catalog tables for all databases in a single dbspace but instead place the system catalog tables with the database tables that they describe.

To create a system catalog table in the table dbspace

1. Create a database in the dbspace in which the table is to reside.
2. Use the SQL statements DATABASE or CONNECT to make that database the current database.
3. Enter the CREATE TABLE statement to create the table.

Placement of Critical Data

The disk or disks that contain the system reserved pages, the physical log, and the dbspaces that contain the logical-log files are critical to the operation of the database server. The database server cannot operate if any of these elements becomes unavailable. By default, the database server places all three critical elements in the root dbspace.

To arrive at an appropriate placement strategy for critical data, you must make a trade-off between the availability of data and maximum logging performance.

The database server also places temporary table and sort files in the root dbspace by default. Informix recommends that you use the DBSPACETEMP configuration parameter and the **DBSPACETEMP** environment variable to assign these tables and files to other dbspaces. For details, see “Configuring Dbspaces for Temporary Tables and Sort Files” on page 5-12.

Consider Separate Disks for Critical Data Components

If you place the root dbspace, logical log, and physical log in separate dbspaces on separate disks, you can obtain some distinct performance advantages. The disks that you use for each critical data component should be on separate controllers. This approach has the following advantages:

- Isolates logging activity from database I/O and allows physical-log I/O requests to be serviced in parallel with logical-log I/O requests
- Reduces the time that you need to recover from a failure

  However, unless the disks are mirrored, there is an increased risk that a disk that contains critical data might be affected in the event of a failure, which will bring the database server to a halt and require the complete restoration of all data from a level-0 backup.
- Allows for a relatively small root dbspace that contains only reserved pages, the database partition, and the **sysmaster** database

  In many cases, 10,000 kilobytes is sufficient.

The database server uses different methods to configure various portions of critical data. To assign an appropriate dbspace for the root dbspace and physical log, set the appropriate database server configuration parameters. To assign the logical-log files to an appropriate dbspace, use the **onparams** utility.

For more information on the configuration parameters that affect each portion of critical data, refer to "Configuration Parameters That Affect Critical Data" on page 5-11.

Consider Mirroring for Critical Data Components

Consider mirroring for the dbspaces that contain critical data. Mirroring these dbspaces ensures that the database server can continue to operate even when a single disk fails. However, depending on the mix of I/O requests for a given dbspace, a trade-off exists between the fault tolerance of mirroring and I/O performance. You obtain a marked performance advantage when you mirror dbspaces that have a read-intensive usage pattern and a slight performance disadvantage when you mirror write-intensive dbspaces.

When mirroring is in effect, two disks are available to handle read requests, and the database server can process a higher volume of those requests. However, each write request requires two physical write operations and does not complete until both physical operations are performed. The write operations are performed in parallel, but the request does not complete until the slower of the two disks performs the update. Thus, you experience a slight performance penalty when you mirror write-intensive dbspaces.

Mirroring the Root Dbspace

You can achieve a certain degree of fault tolerance with a minimum performance penalty if you mirror the root dbspace and restrict its contents to read-only or seldom-accessed tables. When you place update-intensive tables in other, nonmirrored dbspaces, you can use the database server backup-and-restore facilities to perform warm restores of those tables in the event of a disk failure. When the root dbspace is mirrored, the database server remains online to service other transactions while the failed disk is being repaired.

When you mirror the root dbspace, always place the first chunk on a different device than that of the mirror. The MIRRORPATH configuration parameter should have a different value than ROOTPATH.

Mirroring Smart-Large-Object Chunks

An sbspace is a logical storage unit composed of one or more chunks that store *smart large objects*, which consist of CLOB (character large object) or BLOB (binary large object) data. For a more detailed description of sbspaces, refer to your *Administrator's Guide*.

The first chunk of an sbspace contains a special set of pages, called *metadata*, which is used to locate smart large objects in the sbspace. Additional chunks that are added to the sbspace can also have metadata pages if you specify them on the **onspaces** command when you create the chunk.

Consider mirroring chunks that contain metadata pages for the following reasons:

- Higher availability

  Without access to the metadata pages, users cannot access any smart large objects in the sbspace. If the first chunk of the sbspace contains all of the metadata pages and the disk that contains that chunk becomes unavailable, you cannot access a smart large object in the sbspace, even if it resides on a chunk on another disk. For high availability, mirror at least the first chunk of the sbspace and any other chunk that contains metadata pages.

- Faster access

  By mirroring the chunk that contains the metadata pages, you can spread read activity across the disks that contain the primary chunk and mirror chunk.

Mirroring the Logical Log

The logical log is write intensive. If the dbspace that contains the logical-log files is mirrored, you encounter the slight double-write performance penalty noted in "Consider Mirroring for Critical Data Components" on page 5-8. However, you can adjust the rate at which logging generates I/O requests to a certain extent by choosing an appropriate log buffer size and logging mode.

With unbuffered and ANSI-compliant logging, the database server requests a flush of the log buffer to disk for every committed transaction (two when the dbspace is mirrored). Buffered logging generates far fewer I/O requests than unbuffered or ANSI-compliant logging.

With buffered logging, the log buffer is written to disk only when it fills and all the transactions that it contains are completed. You can reduce the frequency of logical-log I/O even more if you increase the size of your logical-log buffers. However, buffered logging leaves transactions in any partially filled buffers vulnerable to loss in the event of a system failure.

Although database consistency is guaranteed under buffered logging, specific transactions are not guaranteed against a failure. The larger the logical-log buffers, the more transactions you might need to reenter when service is restored after a failure.

Unlike the physical log, you cannot specify an alternative dbspace for logical-log files in your initial database server configuration. Instead, use the **onparams** utility first to add logical-log files to an alternative dbspace and then drop logical-log files from the root dbspace. For more information about **onparams**, refer to the *Administrator's Reference*.

Mirroring the Physical Log

The physical log is write intensive, with activity occurring at checkpoints and when buffered data pages are flushed. I/O to the physical log also occurs when a page-cleaner thread is activated. If the dbspace that contains the physical log is mirrored, you encounter the slight double-write performance penalty noted under "Consider Mirroring for Critical Data Components" on page 5-8.

To keep I/O to the physical log at a minimum, you can adjust the checkpoint interval and the LRU minimum and maximum thresholds. (See "CKPINTVL" on page 5-42 and "LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY" on page 5-56.)

Configuration Parameters That Affect Critical Data

You can use the following configuration parameters to configure the root dbspace:

- ROOTNAME
- ROOTOFFSET
- ROOTPATH
- ROOTSIZE
- MIRROR
- MIRRORPATH
- MIRROROFFSET

These parameters determine the location and size of the initial chunk of the root dbspace and configure mirroring, if any, for that chunk. (If the initial chunk is mirrored, all other chunks in the root dbspace must also be mirrored). Otherwise, these parameters have no major impact on performance.

The following configuration parameters affect the logical logs:

- LOGSIZE
- LOGBUFF

LOGSIZE determines the size of each logical-log files. LOGBUFF determines the size of the three logical-log buffers that are in shared memory. For more information on LOGBUFF, refer to "LOGBUFF" on page 4-21.

The following configuration parameters determine the location and size of the physical log:

- PHYSDBS
- PHYSFILE

Configuring Dbspaces for Temporary Tables and Sort Files

Applications that use temporary tables or large sort operations require a large amount of temporary space. To improve performance of these applications, use the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable to designate one or more dbspaces for temporary tables and sort files.

Depending on how the temporary space is created, the database server uses the following default locations for temporary table and sort files when you do not set DBSPACETEMP:

- The dbspace of the current database, when you create an explicit temporary table with the TEMP TABLE clause of the CREATE TABLE statement and do not specify a dbspace for the table either in the IN DBSPACE clause or in the FRAGMENT BY clause

  This action can severely affect I/O to that dbspace. If the root dbspace is mirrored, you encounter a slight double-write performance penalty for I/O to the temporary tables and sort files.

- The root dbspace when you create an explicit temporary table with the INTO TEMP option of the SELECT statement

  This action can severely affect I/O to the root dbspace. If the root dbspace is mirrored, you encounter a slight double-write performance penalty for I/O to the temporary tables and sort files.

- The operating-system directory or file that you specify in one of the following variables:

  - In UNIX, the operating-system directory or directories that the **PSORT_DBTEMP** environment variable specifies, if it is set

    If **PSORT_DBTEMP** is not set, the database server writes sort files to the operating-system file space in the **/tmp** directory.

  - In Windows NT, the directory specified in **TEMP** or **TMP** in the User Environment Variables window on **Control Panel→System.**

  The database server uses the operating-system directory or files to direct any overflow that results from the following database operations:

  - SELECT statement with GROUP BY clause
  - SELECT statement with ORDER BY clause
  - Hash-join operation
  - Nested-loop join operation
  - Index builds

Warning: *If you do not specify a value for the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable, the database server uses this operating-system file for implicit temporary tables. If this file system has insufficient space to hold a sort file, the query performing the sort returns an error. Meanwhile, the operating system might be severely impacted until you remove the sort file.*

You can improve performance with the use of temporary dbspaces that you create exclusively to store temporary tables and sort files. Informix recommends that you use the DBSPACETEMP configuration parameter and the **DBSPACETEMP** environment variable to assign these tables and files to temporary dbspaces.

When you specify dbspaces in either the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable, you gain the following performance advantages:

- Reduced I/O impact on the root dbspace, production dbspaces, or operating-system files
- Use of parallel inserts into the temporary table when you specify more than one dbspace for temporary tables

  The database server automatically applies its parallel insert capability to fragment the temporary table across those dbspaces, using a round-robin distribution scheme.
- Improved speed with which the database server creates temporary tables when you assign two or more temporary dbspaces on separate disks

Important: *Informix recommends that you use the DBSPACETEMP parameter or the* ***DBSPACETEMP*** *environment variable for better performance of sort operations and to prevent the database server from unexpectedly filling file systems. The dbspaces that you list must be composed of chunks that are allocated as unbuffered devices.*

Creating Temporary Dbspaces

To create a dbspace for the exclusive use of temporary tables and sort files, use **onspaces -t**. For best performance, use the following guidelines:

- If you create more than one temporary dbspace, create each dbspace on a separate disk to balance the I/O impact.
- Place no more than one temporary dbspace on a single disk.

The database server does not perform logical or physical logging of temporary dbspaces, and temporary dbspaces are never backed up as part of a full-system backup. You cannot mirror a temporary dbspace that you create with **onspaces -t**.

Important: *In the case of a database with logging, you must include the WITH NO LOG clause in the SELECT... INTO TEMP statement to place the explicit temporary tables in the dbspaces listed in the DBSPACETEMP configuration parameter and the* ***DBSPACETEMP*** *environment variable. Otherwise, the database server stores the explicit temporary tables in the root dbspace.*

For information about how to create temporary dbspaces, see your *Administrator's Guide*.

DBSPACETEMP Configuration Parameter

The DBSPACETEMP configuration parameter specifies a list of dbspaces in which the database server places temporary tables and sort files by default. Some or all of the dbspaces that you list in this configuration parameter can be temporary dbspaces, which are reserved exclusively to store temporary tables and sort files.

If you specify more than one dbspace in this list, the database server uses its parallel insert capability to fragment temporary tables across all the listed dbspaces, using a round-robin distribution scheme. For more information, refer to "Designing a Distribution Scheme" on page 9-11.

The DBSPACETEMP configuration parameter lets the database administrator restrict which dbspaces the database server uses for temporary storage. For detailed information about the settings of DBSPACETEMP, see the *Administrator's Reference*.

Important: *The DBSPACETEMP configuration parameter is not set in the* ***onconfig.std*** *file. For best performance with temporary tables and sort files, Informix recommends that you use DBSPACETEMP to specify two or more dbspaces on separate disks.*

DBSPACETEMP Environment Variable

To override the DBSPACETEMP parameter, you can use the **DBSPACETEMP** environment variable for both temporary tables and sort files. This environment variable specifies a comma- or colon-separated list of dbspaces in which to place temporary tables for the current session.

Important: *Informix recommends that you use the DBSPACETEMP parameter or the* ***DBSPACETEMP*** *environment variable for better performance of sort operations and to prevent the database server from unexpectedly filling file systems.*

Informix recommends that you use **DBSPACETEMP** rather than the **PSORT_DBTEMP** environment variable to specify sort files for the following reasons:

- **DBSPACETEMP** typically yields better performance.

  When dbspaces reside on character-special devices (also known as raw disk devices), the database server uses unbuffered disk access. I/O is faster to unbuffered devices than to regular (buffered) operating-system files because the database server manages the I/O operation directly.
- **PSORT_DBTEMP** specifies one or more operating-system directories in which to place sort files.

  These operating-system files can unexpectedly fill on your computer because the database server does not manage them.

Estimating Temporary Space

You can use the following guidelines to estimate the amount of temporary space to allocate:

- For OLTP applications, allocate temporary dbspaces that equal at least 10 percent of the table.
- For DSS applications, allocate temporary dbspaces that equal at least 50 percent of the table.

Hash joins can use a significant amount of memory and can potentially overflow to temporary space on disk. You can use the following formula to estimate the amount of memory that is required for the hash table in a hash join:

```
hash_table_size = (32 bytes + row_size) * num_rows_smalltab
```

The value for *num_rows_smalltab* should be the number of rows in the probe table. For more information on hash joins, refer to "Hash Join" on page 10-6.

PSORT_NPROCS Environment Variable

The **PSORT_NPROCS** environment variable specifies the maximum number of threads that the database server can use to sort a query. When a query involves a large sort operation, multiple sort threads can execute in parallel to improve the performance of the query.

When the value of PDQ priority is `0` and **PSORT_NPROCS** is greater than 1, the database server uses parallel sorts. The management of PDQ does not limit these sorts. In other words, although the sort is executed in parallel, the database server does not regard sorting as a PDQ activity. When PDQ priority is `0`, the database server does not control sorting by any of the PDQ configuration parameters.

When PDQ priority is greater than 0 and **PSORT_NPROCS** is greater than 1, the query benefits both from parallel sorts and from PDQ features such as parallel scans and additional memory. Users can use the **PDQPRIORITY** environment variable to request a specific proportion of PDQ resources for a query. You can use the MAX_PDQPRIORITY parameter to limit the number of such user requests. For more information on MAX_PDQPRIORITY, refer to "MAX_PDQPRIORITY" on page 3-16.

The database server allocates a relatively small amount of memory for sorting, and that memory is divided among the **PSORT_NPROCS** sort threads. Sort processes use temporary space on disk when not enough memory is allocated. For more information on memory allocated for sorting, refer to "Estimating Sort Memory" on page 7-19.

Important: *For better performance for a sort operation, Informix recommends that you set PSORT_NPROCS initially to 2 if your computer has multiple CPUs. If the subsequent CPU activity is lower than I/O activity, you can increase the value of PSORT_NPROCS.*

For more information on sorts during index builds, refer to "Improving Performance for Index Builds" on page 7-18.

Configuring Sbspaces for Temporary Smart Large Objects

Applications can use temporary smart large objects for text, image, or other user-defined data types that are only required during the life of the user session. These applications do not require logging of the temporary smart large objects. Logging adds I/O activity to the logical log and increases memory utilization.

You can store temporary smart large objects in a permanent sbspace or a temporary sbspace.

- Permanent sbspaces

  If you store the temporary smart large objects in a regular sbspace and keep the default no logging attribute, changes to the objects are not logged, but the metadata is always logged.

- Temporary sbspaces

  Applications that update temporary smart large objects stored in temporary sbspaces are significantly faster because the database server does not log the metadata or the user data in a temporary sbspace.

To improve performance of applications that update temporary smart large objects, specify the **LOTEMP** flag in the **mi_lo_specset_flags or ifx_lo_specset_flags** API function and specify a temporary sbspace for the temporary smart large objects. The database server uses the following order of precedence for locations to place temporary smart large objects:

- The sbspace you specify in the **mi_lo_specset_sbspace** or **ifx_lo_specset_sbspace** API function when you create the smart large object

  Specify a temporary sbspace in the API function so that changes to the objects and the metadata are not logged. The sbspace you specify in the API function overrides any default sbspaces that the SBSPACETEMP or SBSPACENAME configuration parameters might specify.

- The sbspace you specify in the IN SBSPACE clause when you create an explicit temporary table with the TEMP TABLE clause of the CREATE TABLE statement

  Specify a temporary sbspace in the IN SBSPACE clause so that changes to the objects and the metadata are not logged.
- The permanent sbspace you specify in the SBSPACENAME configuration parameter, if you do not specify an sbspace in the SBSPACETEMP configuration parameter

If no temporary sbspace is specified in any of the above methods, then the database server issues the following error message when you try to create a temporary smart large object:

```
-12053  Smart Large Objects: No sbspace number specified.
```

Creating Temporary Sbspaces

To create an sbspace for the exclusive use of temporary smart large objects, use **onspaces -c -S** with the **-t** option. For best performance, use the following guidelines:

- If you create more than one temporary sbspace, create each sbspace on a separate disk to balance the I/O impact.
- Place no more than one temporary sbspace on a single disk.

The database server does not perform logical or physical logging of temporary sbspaces, and temporary sbspaces are never backed up as part of a full-system backup. You cannot mirror a temporary sbspace that you create with **onspaces -t**.

Important: *In the case of a database with logging, you must include the WITH NO LOG clause in the SELECT... INTO TEMP statement to place the temporary smart large objects in the sbspaces listed in the SBSPACETEMP configuration parameter. Otherwise, the database server stores the temporary smart large objects in the sbspace listed in the SBSPACENAME configuration parameter.*

For information about how to create temporary sbspaces, refer to your *Administrator's Guide*. For information about how to create temporary smart large objects, refer to the *DataBlade API Programmer's Manual*.

SBSPACETEMP Configuration Parameter

The SBSPACETEMP configuration parameter specifies a list of sbspaces in which the database server places temporary smart large objects by default. Some or all of the sbspaces that you list in this configuration parameter can be temporary sbspaces, which are reserved exclusively to store temporary smart large objects.

The SBSPACETEMP configuration parameter lets the database administrator restrict which sbspaces the database server uses for temporary storage. For detailed information about the settings of SBSPACETEMP, see the *Administrator's Reference*.

Important: *The SBSPACETEMP configuration parameter is not set in the* ***onconfig.std*** *file. For best performance with temporary smart large objects, Informix recommends that you use SBSPACETEMP to specify two or more sbspaces on separate disks.*

Placement of Simple Large Objects

You can store simple large objects in either the same dbspace in which the table resides or in a blobspace.

A blobspace is a logical storage unit composed of one or more chunks that store only simple large objects (TEXT or BYTE data). For information on sbspaces, which store smart large objects (such as BLOB, CLOB, or multirepresentational data), refer to "Parameters That Affect I/O for Smart Large Objects" on page 5-27.

If you use a blobspace, you can store simple large objects on a separate disk from the table with which the data is associated. You can store simple large objects associated with different tables in the same blobspace.

UNIX

You can create a blobspace with the following utilities:

- ON-Monitor ♦
- **ISA**
- **onspaces**

For details on these utilities, see the *Administrator's Reference*.

You assign simple large objects to a blobspace when you create the tables with which simple large objects are associated. For more information on the SQL statement CREATE TABLE, refer to the *Informix Guide to SQL: Syntax*.

Simple large objects are not logged and do not pass through the buffer pool. However, frequency of checkpoints can affect applications that access TEXT or BYTE data. For more information, refer to "LOGSIZE and LOGFILES" on page 5-43.

Advantage of Blobspaces over Dbspaces

When you store simple large objects in a blobspace on a separate disk from the table with which it is associated, the database server provides the following performance advantages:

- You have parallel access to the table and simple large objects.
- Unlike simple large objects stored in a dbspace, blobspace data is written directly to disk. Simple large objects do not pass through resident shared memory, which leaves memory pages free for other uses.
- Simple large objects are not logged, which reduces logging I/O activity for logged databases.

For more information, refer to "Storing Simple Large Objects in the Tblspace or a Separate Blobspace" on page 6-18.

Blobpage Size Considerations

Blobspaces are divided into units called *blobpages*. The database server retrieves simple large objects from a blobspace in blobpage-sized units.You specify the size of a blobpage in multiples of a disk page when you create the blobspace. The optimal blobpage size for your configuration depends on the following factors:

- The size distribution of the simple large objects
- The trade-off between retrieval speed for your largest simple large object and the amount of disk space that is wasted by storing simple large objects in large blobpages

To retrieve simple large objects as quickly as possible, use the size of your largest simple large object rounded up to the nearest disk-page-sized increment. This scheme guarantees that the database server can retrieve even the largest simple large object in a single I/O request. Although this scheme guarantees the fastest retrieval, it has the potential to waste disk space. Because simple large objects are stored in their own blobpage (or set of blobpages), the database server reserves the same amount of disk space for every blobpage even if the simple large object takes up a fraction of that page. Using a smaller blobpage allows you to make better use of your disk, especially when large differences exist in the sizes of your simple large objects.

To achieve the greatest theoretical utilization of space on your disk, you could make your blobpage the same size as a standard disk page. Then many, if not most, simple large objects would require several blobpages. Because the database server acquires a lock and issues a separate I/O request for each blobpage, this scheme performs poorly.

In practice, a balanced scheme for sizing uses the most frequently occurring simple-large-object size as the size of a blobpage. For example, suppose that you have 160 simple-large-object values in a table with the following size distribution:

- Of these values, 120 are 12 kilobytes each.
- The other 40 values are 16 kilobytes each.

You can choose one of the following blobpage sizes:

- The 12-kilobyte blobpage size provides greater storage efficiency than a 16-kilobyte blobpage size, as the following two calculations show:
  - 12 kilobytes

    This configuration allows the majority of simple-large-object values to require a single blobpage and the other 40 values to require two blobpages. In this configuration, 8 kilobytes is wasted in the second blobpage for each of the larger values. The total wasted space is as follows:

    ```
    wasted-space = 8 kilobtyes * 40
                 = 329 kilobytes
    ```

  - 16 kilobytes

    In this configuration, 4 kilobytes is wasted in the extents of 120 simple large objects. The total wasted space is as follows:

    ```
    wasted-space = 4 kilobtyes * 120
                 = 640 kilobytes
    ```
- If your applications access the 16-kilobyte simple-large-object values more frequently, the database server must perform a separate I/O operation for each blobpage. In this case, the 16-kilobyte blobpage size provides better retrieval speed than a 12-kilobyte blobpage size.

Tip: *If a table has more than one simple-large-object column and the data values are not close in size, store the data in different blobspaces, each with an appropriately sized blobpage.*

Optimizing Blobspace Blobpage Size

When you are evaluating blobspace storage strategy, you can measure efficiency by two criteria:

- Blobpage fullness
- Blobpages required per simple large object

Blobpage fullness refers to the amount of data within each blobpage. TEXT and BYTE data stored in a blobspace cannot share blobpages. Therefore, if a single simple large object requires only 20 percent of a blobpage, the remaining 80 percent of the page is unavailable for use. However, avoid making the blobpages too small. When several blobpages are needed to store each simple large object, you increase the overhead cost of storage. For example, more locks are required for updates, because a lock must be acquired for each blobpage.

Obtaining Blobspace Storage Statistics

To help you determine the optimal blobpage size for each blobspace, use the **oncheck -pB** command. The **oncheck -pB** command lists the following statistics for each table (or database):

- The number of blobpages used by the table (or database) in each blobspace
- The average fullness of the blobpages used by each simple large object stored as part of the table (or database)

Determining Blobpage Fullness with oncheck -pB

The **oncheck -pB** command displays statistics that describe the average fullness of blobpages. These statistics provide a measure of storage efficiency for individual simple large objects in a database or table. If you find that the statistics for a significant number of simple large objects show a low percentage of fullness, the database server might benefit from changing the size of the blobpage in the blobspace.

Both the **oncheck -pB** and **onstat -d update** outputs display the same information about the number of free blobpages. For information about **onstat -d update**, see managing disk space in the *Administrator's Guide*.

Execute **oncheck -pB** with either a database name or a table name as a parameter. The following example retrieves storage information for all simple large objects stored in the table **sriram.catalog** in the **stores_demo** database:

```
oncheck -pB stores_demo:sriram.catalog
```

Figure 5-1 shows the output of this command.

```
                BLOBSpace Report for stores_demo:sriram.catalog

Total pages used by table7

BLOBSpace usage:
Space   Page             Percent Full
Name    Number    Pages  0-25%  26-50%  51-75  76-100%
-------------------------------------------------------------
blobPIC 0x300080  1      x
   blobPIC 0x300082  2      x
         ------
Page Size is 61443

bspc1   0x2000b2  2             x
bspc1   0x2000b6  2                     x
         ------
Page Size is 20484
```

Figure 5-1
Output of oncheck -pB

Space Name is the name of the blobspace that contains one or more simple large objects stored as part of the table (or database).

Page Number is the starting address in the blobspace of a specific simple large object.

Pages is the number of the database server pages required to store this simple large object.

Percent Full is a measure of the average blobpage fullness, by blobspace, for each blobspace in this table or database.

Page Size is the size in bytes of the blobpage for this blobspace. Blobpage size is always a multiple of the database server page size.

The example output indicates that four simple large objects are stored as part of the table **sriram.catalog**. Two objects are stored in the blobspace **blobPIC** in 6144-byte blobpages. Two more objects are stored in the blobspace **bspc1** in 2048-byte blobpages.

The summary information that appears at the top of the display, **Total pages used by table** is a simple total of the blobpages needed to store simple large objects. The total says nothing about the size of the blobpages used, the number of simple large objects stored, or the total number of bytes stored.

The efficiency information displayed under the **Percent Full** heading is imprecise, but it can alert an administrator to trends in the storage of TEXT and BYTE data.

Interpreting Blobpage Average Fullness

This section demonstrates the idea of average fullness. The first simple large object listed in Figure 5-1 on page 5-25 is stored in the blobspace **blobPIC** and requires one 6144-byte blobpage. The blobpage is 51 to 75 percent full, meaning that the size is between 0.51 * 6144 = 3133 bytes and 0.75 * 6144 = 4608. The maximum size of this simple large object must be less than or equal to 75 percent of 6144 bytes, or 4608 bytes.

The second object listed under blobspace **blobPIC** requires two 6144-byte blobpages for storage, or a total of 12,288 bytes. The average fullness of all allocated blobpages is 51 to 75 percent. Therefore, the minimum size of the object must be greater than 50 percent of 12,288 bytes, or 6144 bytes. The maximum size of the simple large object must be less than or equal to 75 percent of 12,288 bytes, or 9216 bytes. The average fullness does not mean that each page is 51 to 75 percent full. A calculation would yield 51 to 75 percent average fullness for two blobpages where the first blobpage is 100 percent full and the second blobpage is 2 to 50 percent full.

Now consider the two simple large objects in blobspace **bspc1**. These two objects appear to be nearly the same size. Both objects require two 2048-byte blobpages, and the average fullness for each is 76 to 100 percent. The minimum size for these simple large objects must be greater than 75 percent of the allocated blobpages, or 3072 bytes. The maximum size for each object is slightly less than 4096 bytes (allowing for overhead).

Applying Efficiency Criteria to Output

Looking at the efficiency information for blobspace **bspc1**, a database server administrator might decide that a better storage strategy for TEXT and BYTE data would be to double the blobpage size from 2048 bytes to 4096 bytes. (Blobpage size is always a multiple of the database server page size.) If the database server administrator made this change, the measure of page fullness would remain the same, but the number of locks needed during an update of a simple large object would be reduced by half.

The efficiency information for blobspace **blobPIC** reveals no obvious suggestion for improvement. The two simple large objects in **blobPIC** differ considerably in size, and there is no optimal storage strategy. In general, simple large objects of similar size can be stored more efficiently than simple large objects of different sizes.

IDS

Parameters That Affect I/O for Smart Large Objects

An sbspace is a logical storage unit, composed of one or more chunks, in which you can store smart large objects (such as BLOB, CLOB, or multi representational data). For more information about sbspaces, refer to the chapter on where data is stored in your *Administrator's Guide*.

This section provides guidelines for the following topics:

- Disk layout
- Configuration parameters
- Options for the **onspaces** utility

The DataBlade API and the ESQL/C application programming interface also provide functions that affect I/O operations for smart large objects.

Important: *For most applications, Informix recommends that you use the values that the database server calculates for the disk-storage information. For more information, see the "Informix ESQL/C Programmer's Manual" and the "DataBlade API Programmer's Manual."*

Disk Layout for Sbspaces

You create sbspaces on separate disks from the table with which the data is associated. You can store smart large objects associated with different tables within the same sbspace.

When you store smart large objects in an sbspace on a separate disk from the table with which it is associated, the database server provides the following performance advantages:

- You have parallel access to the table and smart large objects.
- When you choose not to log the data in an sbspace, you reduce logging I/O activity for logged databases.

To create an sbspace, use the **onspaces** utility, as your *Administrator's Reference* describes. You assign smart large objects to an sbspace when you create the tables with which the smart large objects are associated. For more information on the SQL statement CREATE TABLE, refer to the *Informix Guide to SQL: Syntax*.

Configuration Parameters That Affect Sbspace I/O

The following configuration parameters affect the I/O performance of sbspaces:

- SBSPACENAME
- BUFFERS
- LOGBUFF

SBSPACENAME

The SBSPACENAME configuration parameter indicates the default sbspace name if you do not specify the sbspace name when you define a column of data type CLOB or BLOB. To reduce disk contention and provide better load balancing, place the default sbspace on a separate disk from the table data.

BUFFERS

The size of your memory buffer pool affects I/O operations for smart large objects because the buffer pool is the default area of shared memory for these objects. If your applications frequently access smart large objects that are 2 kilobytes or 4 kilobytes in size, use the buffer pool to keep them in memory longer. For information on estimating the amount to increase your buffer pool for smart large objects, refer to "Smart Large Objects and BUFFERS" on page 4-16.

By default, the database server reads smart large objects into the buffers in the resident portion of shared memory. For more information on using light-weight I/O buffers, refer to "Lightweight I/O for Smart Large Objects" on page 5-32.

LOGBUFF

The LOGBUFF parameter affects logging I/O activity because it specifies the size of the logical-log buffers that are in shared memory. The size of these buffers determines how quickly they fill and therefore how often they need to be flushed to disk.

If you log smart-large-object user data, increase the size of your logical-log buffer to prevent frequent flushing to these log files on disk.

The user-data portion of smart large objects that are logged does not pass through the physical log, so you do not need to adjust the PHYSBUF parameter for smart large objects.

onspaces Options That Affect Sbspace I/O

When you create an sbspace, you can specify the following **onspaces** options that affect I/O performance:

- Extent sizes
- Buffering mode (use lightweight I/O or not)
- Logging

Sbspace Extent Sizes

As you add smart large objects to a table, the database server allocates disk space to the sbspace in units called *extents*. Each extent is a block of physically contiguous pages from the sbspace. Even when the sbspace includes more than one chunk, each extent is allocated entirely within a single chunk so that it remains contiguous.

Contiguity is important to I/O performance. When the pages of data are contiguous, disk-arm motion is minimized when the database server reads the rows sequentially. The mechanism of extents is a compromise between the following competing requirements:

- The size of some smart large objects is not known in advance.
- The number of smart large objects in different tables can grow at different times and different rates.
- All the pages of a single smart large object should ideally be adjacent for best performance when you retrieve the entire object.

Because you might not be able to predict the number and size of smart large objects, you cannot specify the extent length of smart large objects. Therefore, the database server adds extents only as they are needed, but all the pages in any one extent are contiguous for better performance. In addition, when the database server creates a new extent that is adjacent to the previous extent, it treats both extents as a single extent.

The number of pages in an sbspace extent is determined by one of the following methods:

- The database server calculates the extent size for a smart large object from a set of heuristics, such as the number of bytes in a write operation. For example, if an operation asks to write 30 kilobytes, the database server tries to allocate an extent the size of 30 kilobytes.

- The final size of the smart large object as indicated by one of the following functions when you open the sbspace in an application program:

  DB API

  - The DataBlade API **mi_lo_specset_estbytes** function

    For more information on the DataBlade API functions to open a smart large object and set the estimated number of bytes, see the *DataBlade API Programmer's Manual.* ♦

  E/C

  - The ESQL/C **ifx_lo_specset_estbytes** function

    For more information on the ESQL/C functions to open a smart large object and set the estimated number of bytes, see the *Informix ESQL/C Programmer's Manual.* ♦

  These functions are the best way to set the extent size because they reduce the number of extents in a smart large object. The database server tries to allocate the entire smart large object as one extent (if an extent of that size is available in the chunk).

- The EXTENT_SIZE flag in the **-Df** option of the **onspaces** command when you create or alter the sbspace

  Most administrators do not use the **onspaces** EXTENT_SIZE flag because the database server calculates the extent size from heuristics. However, you might consider using the **onspaces** EXTENT_SIZE flag in the following situations:

  - Many one-page extents are scattered throughout the sbspace.
  - Almost all smart large objects are the same length.

- The EXTENT SIZE keyword of the CREATE TABLE statement when you define the CLOB or BLOB column

  Most administrators do not use the EXTENT SIZE keyword when they create or alter a table because the database server calculates the extent size from heuristics. However, you might consider using this EXTENT SIZE keyword if almost all smart large objects are the same length.

Important: *For most applications, Informix recommends that you use the values that the database server calculates for the extent size. Do not use the DataBlade API* ***mi_lo_specset_extsz*** *function or the ESQL/C* ***ifx_lo_specset_extsz*** *function to set the extent size of the smart large object.*

If you know the size of the smart large object, Informix recommends that you specify the size in the DataBlade API **mi_lo_specset_estbytes()** function or ESQL/C **ifx_lo_specset_estbytes()** function instead of in the **onspaces** utility or the CREATE TABLE or the ALTER TABLE statement. These functions are the best way to set the extent size because the database server allocates the entire smart large object as one extent (if it has contiguous storage in the chunk).

Extent sizes over one megabyte do not provide much I/O benefit because the database server performs read and write operations in multiples of 60 kilobytes at the most. However, the database server registers each extent for a smart large object in the metadata area; therefore, especially large smart large objects might have many extent entries. Performance of the database server might degrade when it accesses these extent entries. In this case, you can reduce the number of extent entries in the metadata area if you specify the eventual size of the smart large object in the **mi_lo_specset_estbytes()** function or **ifx_lo_specset_estbytes()** function.

For more information, refer to "Improving Metadata I/O for Smart Large Objects" on page 6-23.

Lightweight I/O for Smart Large Objects

By default, smart large objects pass through the buffer pool in the resident portion of shared memory. Although smart large objects have lower priority than other data, the buffer pool can become full when an application accesses many smart large objects. A single application can fill the buffer pool with smart large objects and leave little room for data that other applications might need. In addition, when the database server performs scans of many pages into the buffer pool, the overhead and contention associated with checking individual pages in and out might become a bottleneck.

Instead of using the buffer pool, the administrator and programmer have the option to use *lightweight I/O*. Lightweight I/O operations use private buffers in the session pool of the virtual portion of shared memory.

Important: *Use private buffers only when you read or write smart large objects in read or write operations greater than 8080 bytes and you seldom access them. That is, if you have infrequent read or write function calls that read large amounts of data in a single function invocation, lightweight I/O can improve I/O performance.*

Advantages of Lightweight I/O for Smart Large Objects

Lightweight I/O provides the following advantages:

- Transfers larger blocks of data in one I/O operation

  These I/O blocks can be as large as 60 kilobytes. But the bytes must be adjacent for the database server to transfer them in a single I/O operation.
- Bypasses the overhead of the buffer pool when many pages are read
- Prevents frequently accessed pages from being forced out of the buffer pool when many sequential pages are read for smart large objects

When you use lightweight I/O buffers for smart large objects, the database server might read several pages with one I/O operation. A single I/O operation reads in several smart-large-object pages, up to the size of an extent. For information on when to specify extent size, refer to "Sbspace Extent Sizes" on page 5-30.

Specifying Lightweight I/O for Smart Large Objects

To specify the use of lightweight I/O when creating the sbspace, the administrator can use the BUFFERING tag of the **-Df** option in the **onspaces -c -S** command. The default value for BUFFERING is `ON`, which means to use the buffer pool. The buffering mode that you specify (or the default, if you do not specify) in the **onspaces** command is the default buffering mode for all smart large objects stored within the sbspace.

Important: *In general, if read and write operations to the smart large objects are less than 8080 bytes, do not specify a buffering mode when you create the sbspace. If you are reading or writing short blocks of data, such as 2 kilobytes or 4 kilobytes, leave the default of "BUFFERING=ON" to obtain better performance.*

Programmers can override the default buffering mode when they create, open, or alter a smart-large-object instance with DataBlade API and the ESQL/C functions. The DataBlade API and the ESQL/C application programming interface provide the LO_NOBUFFER flag to allow lightweight I/O for smart large objects.

Important: *Use the LO_NOBUFFER flag only when you read or write smart large objects in operations greater than 8080 bytes and you seldom access them. That is, if you have infrequent read or write function calls that read large amounts of data in a single function invocation, lightweight I/O can improve I/O performance.*

For more information on these flags and functions, refer to the *DataBlade API Programmer's Manual* and the *Informix ESQL/C Programmer's Manual*.

Logging

If you decide to log all write operations on data stored in sbspaces, logical-log I/O activity and memory utilization increases. For more information, refer to "LOGBUFF" on page 5-29.

How the Optical Subsystem Affects Performance

The Optical Subsystem extends the storage capabilities of the database server for simple large objects (TEXT or BYTE data) to write-once-read-many (WORM) optical subsystems. The database server uses a cache in memory to buffer initial TEXT or BYTE data pages requested from the Optical Subsystem. The memory cache is a common storage area. The database server adds simple large objects requested by any application to the memory cache as long as the cache has space. To free space in the memory cache, the application must release the TEXT or BYTE data that it is using.

A significant performance advantage occurs when you retrieve TEXT or BYTE data directly into memory instead of buffering that data on disk. Therefore, proper cache sizing is important when you use the Optical Subsystem. You specify the total amount of space available in the memory cache with the OPCACHEMAX configuration parameter. Applications indicate that they require access to a portion of the memory cache when they set the **INFORMIXOPCACHE** environment variable. For details, refer to "INFORMIXOPCACHE" on page 5-37.

Simple large objects that cannot fit entirely into the space that remains in the cache are stored in the blobspace that the STAGEBLOB configuration parameter names. This staging area acts as a secondary cache on disk for blobpages that are retrieved from the Optical Subsystem. Simple large objects that are retrieved from the Optical Subsystem are held in the staging area until the transactions that requested them are complete.

The database server administrator creates the staging-area blobspace in one of the following ways:

UNIX

- With ON-Monitor ♦
- With the ISA utility
- With the **onspaces** utility

You can use **onstat -O** or ISA (**Performance→Cache→Optical Cache)** to monitor utilization of the memory cache and STAGEBLOB blobspace. If contention develops for the memory cache, increase the value listed in the configuration file for OPCACHEMAX. (The new value takes effect the next time that the database server initializes shared memory.) For a complete description of the Optical Subsystem, refer to the *Guide to the Optical Subsystem*.

Environment Variables and Configuration Parameters for the Optical Subsystem

The following configuration parameters affect the performance of the Optical Subsystem:

- STAGEBLOB
- OPCACHEMAX

The following sections describe these parameters, along with the **INFORMIXOPCACHE** environment variable.

STAGEBLOB

The STAGEBLOB configuration parameter identifies the blobspace that is to be used as a staging area for TEXT or BYTE data that is retrieved from the Optical Subsystem, and it activates the Optical Subsystem. If the configuration file does not list the STAGEBLOB parameter, the Optical Subsystem does not recognize the optical-storage subsystem.

The structure of the staging-area blobspace is the same as all other database server blobspaces. When the database server administrator creates the staging area, it consists of only one chunk, but you can add more chunks as desired. You cannot mirror the staging-area blobspace. The optimal size for the staging-area blobspace depends on the following factors:

- The frequency of simple-large-object storage
- The frequency of simple-large-object retrieval
- The average size of the simple large object to be stored

To calculate the size of the staging-area blobspace, you must estimate the number of simple large objects that you expect to reside there simultaneously and multiply that number by the average simple-large-object size.

OPCACHEMAX

The OPCACHEMAX configuration parameter specifies the total amount of space that is available for simple-large-object retrieval in the memory cache that the Optical Subsystem uses. Until the memory cache fills, it stores simple large objects that are requested by any application. Simple large objects that cannot fit in the cache are stored on disk in the blobspace that the STAGEBLOB configuration parameter indicates. You can increase the size of the cache to reduce contention among simple-large-object requests and to improve performance for requests that involve the Optical Subsystem.

INFORMIXOPCACHE

The **INFORMIXOPCACHE** environment variable sets the size of the memory cache that a given application uses for simple-large-object retrieval. If the value of this variable exceeds the maximum that the OPCACHEMAX configuration parameter specifies, OPCACHEMAX is used instead. If **INFORMIXOPCACHE** is not set in the environment, the cache size is set to OPCACHEMAX by default.

Table I/O

One of the most frequent functions that the database server performs is to bring data and index pages from disk into memory. Pages can be read individually for brief transactions and sequentially for some queries. You can configure the number of pages that the database server brings into memory and the timing of I/O requests for sequential scans. You can also indicate how the database server is to respond when a query requests data from a dbspace that is temporarily unavailable. The following sections describe these methods of reading pages.

For information about I/O for smart large objects, refer to "Parameters That Affect I/O for Smart Large Objects" on page 5-27.

Sequential Scans

When the database server performs a sequential scan of data or index pages, most of the I/O wait time is caused by seeking the appropriate starting page. To dramatically improve performance for sequential scans, you can bring in a number of contiguous pages with each I/O operation. The action of bringing additional pages along with the first page in a sequential scan is called *read-ahead*.

The timing of I/O operations that are needed for a sequential scan is also important. If the scan thread must wait for the next set of pages to be brought in after working its way through each batch, a delay results. Timing second and subsequent read requests to bring in pages before they are needed provides the greatest efficiency for sequential scans. The number of pages to bring in and the frequency of read-ahead I/O requests depends on the availability of space in the memory buffers. Read-ahead can increase page cleaning to unacceptable levels if too many pages are brought in with each batch or if batches are brought in too often. For information on how to configure read-ahead, refer to "RA_PAGES and RA_THRESHOLD" on page 5-40.

Light Scans

In some circumstances, the database server can bypass the overhead of the buffer pool when it performs a sequential scan. Such a sequential scan is termed a *light scan*.

Performance advantages of using light scans instead of the buffer pool for sequential scans are as follows:

- Transfers larger blocks of data in one I/O operation

  These larger I/O blocks are usually 64 kilobytes or 128 kilobytes. To determine the I/O block size that your platform supports, refer to your machine notes file.
- Bypasses the overhead of the buffer pool when many pages are read
- Prevents frequently accessed pages from being forced out of the buffer pool when many sequential pages are read for a single DSS query

Light scans can be used only for sequential scans of large data tables and are the fastest means for performing these scans. System catalog tables, tables smaller than the size of the buffer pool, and tables containing VARCHAR data do not use light scans.

Light scans occur under the following conditions:

- The optimizer chooses a sequential scan of the table.
- The number of pages in the table is greater than the number of buffers in the buffer pool.
- The isolation level obtains no lock or a shared lock on the table:
  - Dirty Read (including nonlogging databases) isolation level
  - Repeatable Read isolation level if the table has a shared or exclusive lock
  - Committed Read isolation if the table has a shared lock

Light scans do not occur under Cursor Stability isolation.

The **onstat -g lsc** output indicates when light scans occur, as the following sample output shows. The **onstat -g lsc** command displays only currently active light scans.

```
Light Scan Info
descriptor  address  next_lpage  next_ppage  ppage_left  bufcnt look_aside
6           aaa7870 3f4         200429      1488        1      N
```

Unavailable Data

Another aspect of table I/O has to do with situations in which a query requests access to a table or fragment in a dbspace that is temporarily unavailable. When the database server determines that a dbspace is unavailable as the result of a disk failure, queries directed to that dbspace fail by default. The database server allows you to specify dbspaces that, when unavailable, can be skipped by queries, as described in "DATASKIP" on page 5-41.

Warning: *If a dbspace containing data that a query requests is listed in DATASKIP and is currently unavailable because of a disk failure, the data that the database server returns to the query can be inconsistent with the actual contents of the database.*

Configuration Parameters That Affect Table I/O

The following configuration parameters affect read-ahead:

- RA_PAGES
- RA_THRESHOLD

In addition, the DATASKIP configuration parameter enables or disables data skipping.

The following sections describe the performance effects and considerations that are associated with these parameters. For more information about database server configuration parameters, refer to the *Administrator's Reference*.

RA_PAGES and RA_THRESHOLD

The RA_PAGES configuration parameter indicates the number of pages that the database server brings into memory in a single I/O operation during sequential scans of data or index pages. The RA_THRESHOLD parameter indicates the point at which the database server issues an I/O request to bring in the next set of pages from disk. Because the greater portion of I/O wait time is involved in seeking the correct starting point on disk, you can increase efficiency of sequential scans by increasing the number of contiguous pages brought in with each transfer. However, setting RA_PAGES too large or RA_THRESHOLD too high with respect to BUFFERS can trigger unnecessary page cleaning to make room for pages that are not needed immediately.

Use the following formulas to calculate values for RA_PAGES and RA_THRESHOLD:

```
RA_PAGES = ((BUFFERS * bp_fract) / (2 * large_queries)) + 2
RA_THRESHOLD = ((BUFFERS * bp_fract) / (2 * large_queries)) - 2
```

bp_fract is the portion of data buffers to use for large scans that require read-ahead. If you want to allow large scans to take up to 75 percent of buffers, *bp_fract* would be `0.75`.

large_queries is the number of concurrent queries that require read-ahead that you intend to support.

DATASKIP

The DATASKIP configuration parameter allows you to specify which dbspaces, if any, queries can skip when those dbspaces are unavailable as the result of a disk failure. You can list specific dbspaces and turn data skipping on or off for all dbspaces. For details, refer to your *Administrator's Guide*.

The database server sets the sixth character in the SQLWARN array to `W` when data skipping is enabled. For more information about the SQLWARN array, refer to the *Informix Guide to SQL: Tutorial*.

Warning: *The database server cannot determine whether the results of a query are consistent when a dbspace is skipped. If the dbspace contains a table fragment, the user who executes the query must ensure that the rows within that fragment are not needed for an accurate query result. Turning DATASKIP on allows queries with incomplete data to return results that can be inconsistent with the actual state of the database. Without proper care, that data can yield incorrect or misleading query results.*

Background I/O Activities

Background I/O activities do not service SQL requests directly. Many of these activities are essential to maintain database consistency and other aspects of database server operation. However, they create overhead in the CPU and take up I/O bandwidth. These overhead activities take time away from queries and transactions. If you do not configure background I/O activities properly, too much overhead for these activities can limit the transaction throughput of your application.

The following list shows some background I/O activities:

- Checkpoints
- Logging
- Page cleaning
- Backup and restore
- Rollback and recovery
- Data replication
- Auditing

Checkpoints occur regardless of whether much database activity occurs, although they can occur with greater frequency as activity increases. Other background activities, such as logging and page cleaning, occur more frequently as database use increases. Activities such as backups, restores, or fast recoveries occur only as scheduled or under exceptional circumstances.

For the most part, tuning your background I/O activities involves striking a balance between appropriate checkpoint intervals, logging modes and log sizes, and page-cleaning thresholds. The thresholds and intervals that trigger background I/O activity often interact; adjustments to one threshold might shift the performance bottleneck to another.

The following sections describe the performance effects and considerations that are associated with the parameters that affect these background I/O activities. For more information about database server configuration parameters, refer to the configuration chapter of the *Administrator's Reference*.

Configuration Parameters That Affect Checkpoints

The following configuration parameters affect checkpoints:

- CKPTINTVL
- LOGSIZE
- LOGFILES
- PHYSFILE
- ONDBSPDOWN

CKPINTVL

The CKPTINTVL configuration parameter specifies the maximum interval between fuzzy checkpoints. As your *Administrator's Guide* describes, the database server takes either a full checkpoint or a fuzzy checkpoint:

- Fuzzy checkpoints take less time to complete than a full checkpoint because the database server flushes fewer pages to disk. Because fuzzy checkpoints take less time to complete, the database server can spend more time processing queries and transactions.
- Full checkpoints flush all dirty pages in the buffer pool to disk. The database server can skip a checkpoint if all data is physically consistent when the checkpoint interval expires.

The database server writes a message to the message log to note the time that it completes a checkpoint. To read these messages, use **onstat -m**.

Checkpoints also occur whenever the physical log becomes 75 percent full. If you set CKPTINTVL to a long interval, you can use physical-log capacity to trigger checkpoints based on actual database activity instead of an arbitrary time unit. However, a long checkpoint interval can increase the time needed for recovery in the event of a failure. Depending on your throughput and data-availability requirements, you can choose an initial checkpoint interval of 5, 10, or 15 minutes, with the understanding that checkpoints might occur more often, depending on physical-logging activity.

LOGSIZE and LOGFILES

The LOGSIZE and LOGFILES configuration parameters indirectly affect checkpoints because they specify the size and number of logical-log files. A checkpoint can occur when the database server detects that the next logical-log file to become current contains the most-recent checkpoint record. If you need to free the logical-log file that contains the last checkpoint, the database server must write a new checkpoint record to the current logical-log file. So if the frequency with which logical-log files are backed up and freed increases, the frequency at which checkpoints occur increases. Although checkpoints block user processing, they no longer last as long. Because other factors (such as the physical-log size) also determine the checkpoint frequency, this effect might not be significant.

When the dynamic log allocation feature is on, the size of the logical log does not affect the thresholds for long transactions as much as it did in previous versions of the database server. For details, refer to "LTXHWM and LTXEHWM" on page 5-54.

The LOGSIZE, LOGFILES, and LOGBUFF parameters also affect logging I/O activity and logical backups. For more information, see "Configuration Parameters That Affect Logging" on page 5-48.

PHYSFILE

The PHYSFILE configuration parameter specifies the size of the physical log. This parameter indirectly affects checkpoints because whenever the physical log becomes 75 percent full, a checkpoint occurs.

Estimating the Size of the Physical Log

If your workload is update intensive and updates tend not to occur on the same pages, you can use the following formula to calculate the size of the physical log where PHYSFILE equals the physical-log size:

```
PHYSFILE = (users * max_log_pages_per_crit_sect * 4 * pagesize) / 1024
```

Variable in Formula	Description
users	The maximum number of concurrent user threads for which you can obtain an estimate when you execute **onstat -u** during peak processing The last line of the **onstat -u** output contains the maximum number of concurrent user threads. If you set the NETTYPE parameter, sum the values specified in the users field of each NETTYPE parameter in your ONCONFIG file.
max_log_pages_per_crit_sect	Maximum number of pages that the database server can physically log in a critical section Substitute one of the following values: ■ 5 if you do not use R-tree indexes ■ 10 if you use R-tree indexes

Variable in Formula	Description
4	Necessary factor because the following part of the formula represents only 25 percent of the physical log: *users * max_log_pages_per_crit_sec*
pagesize	System page size in bytes, which you can obtain with **onstat -b** The **buffer size** field in the last line of this output displays the page size.
1024	Necessary divisor because you specify PHYSFILE parameter in units of kilobytes

This formula is based on how much physical logging space the database server needs in a worst-case scenario. This scenario takes place when a checkpoint occurs because the log becomes 75 percent full. If all the update threads are in a critical section and perform physical logging of the maximum number of pages in their critical section, the database server must fit this logging into the final 25 percent of the physical log to prevent a physical-log overflow.

The exception to this rule occurs if you are using simple large objects in a dbspace in a database without logging. In that case, substitute the size of the most-frequently occurring simple large object in the dbspace for the maximum log pages per critical section.

Understanding How Checkpoints Affect the Physical Log

Also consider the following issues:

- How quickly the physical log fills

  Operations that do not perform updates do not generate before-images. If the size of the database is growing, but applications rarely update the data, not much physical logging occurs, so you might not need a very big physical log.

  Fuzzy checkpoints keep the physical log from filling up too quickly when applications are doing intensive updates. Fuzzy operations do not generate before-images and include inserts, updates, and deletes for rows that contain built-in data types. You can decrease the size of the physical log if you intend to use physical-log fullness to trigger checkpoints or when applications require less intensive updates or when updates tend to cluster within the same pages.

  The database server writes the before-image of only the first nonfuzzy update made to a page. Nonfuzzy updates include the following operations:

  - Inserts, updates, and deletes for rows that contain user-defined data types (UDTs), smart large objects, and simple large objects
  - ALTER statements
  - Operations that create or modify indexes (B-tree, R-tree, or user-defined indexes)

  Thus, you can define a smaller physical log in the following situations:

  - If your application performs mainly fuzzy updates instead of a lot of nonfuzzy updates
  - If your application updates the same pages

- How frequently checkpoints occur

  Because the physical log is emptied after each checkpoint, the physical log only needs to be large enough to hold before-images from changes between checkpoints. If your physical log frequently approaches full, you might consider decreasing the checkpoint interval, CKPTINTVL, so that checkpoints occur more frequently. However, decreasing the checkpoint interval beyond a certain point has an impact on performance.

- When to increase the size of the physical log

  If you increase the checkpoint interval or if you anticipate increased nonfuzzy update activity, you will probably want to increase the size of the physical log.

For information on PHYSFILE and CKPTINTVL, see the chapter on configuration parameters in the *Administrator's Reference*. For information on the physical log and fuzzy checkpoints, see your *Administrator's Guide*.

ONDBSPDOWN

The ONDBSPDOWN configuration parameter specifies the response that the database server makes when an I/O error indicates that a dbspace is down. By default, the database server marks any dbspace that contains no critical data as down and continues processing. Critical data includes the root dbspace, the logical log, or the physical log. To restore access to that database, you must back up all logical logs and then perform a warm restore on the down dbspace.

The database server halts operation whenever a disabling I/O error occurs on a nonmirrored dbspace that contains critical data, regardless of the setting for ONDBSPDOWN. In such an event, you must perform a cold restore of the database server to resume normal database operations.

When ONDBSPDOWN is set to 2, the database server continues processing to the next checkpoint and then suspends processing of all update requests. The database server repeatedly retries the I/O request that produced the error until the dbspace is repaired and the request completes or the database server administrator intervenes. The administrator can use **onmode -O** to mark the dbspace down and continue processing while the dbspace remains unavailable or use **onmode -k** to halt the database server.

Important: *This 2 setting for ONDBSPDOWN can affect the performance for update requests severely because they are suspended due to a down dbspace. When you use this setting for ONDBSPDOWN, be sure to monitor the status of the dbspaces.*

When you set ONDBSPDOWN to 1, the database server treats all dbspaces as though they were critical. Any nonmirrored dbspace that becomes disabled halts normal processing and requires a cold restore. The performance impact of halting and performing a cold restore when any dbspace goes down can be severe.

Important: *If you decide to set ONDBSPDOWN to 1, consider mirroring all your dbspaces.*

Configuration Parameters That Affect Logging

Checkpoints, logging, and page cleaning are necessary to maintain database consistency. A direct trade-off exists between the frequency of checkpoints or the size of the logical logs and the time that it takes to recover the database in the event of a failure. So a major consideration when you attempt to reduce the overhead for these activities is the delay that you can accept during recovery.

The following configuration parameters affect logging:

- LOGBUFF
- PHYSBUFF
- LOGFILES
- LOGSIZE
- DYNAMIC_ LOGS
- LTXHWM
- LTXEHWM

LOGBUFF and PHYSBUFF

The LOGBUFF and PHYSBUFF configuration parameters affect logging I/O activity because they specify the respective sizes of the logical-log and physical-log buffers that are in shared memory. The size of these buffers determines how quickly they fill and therefore how often they need to be flushed to disk.

LOGFILES

The LOGFILES parameter specifies the number of logical-log files.

Determining the Number of Logical Log Files

If all your logical-log files are the same size, you can calculate the total space allocated to the logical-log files as follows:

```
total logical log space = LOGFILES * LOGSIZE
```

If you add logical-log files that are not the size specified by LOGSIZE, you cannot use the `LOGFILES * LOGSIZE` expression to calculate the size of the logical log. Instead, you need to add the sizes for each individual log file on disk.

Use the **onstat -l** utility to monitor logical-log files.

LOGSIZE

Use the LOGSIZE parameter to set the size of each logical log file. It is difficult to predict how much logical-log space your database server system requires until it is fully in use.

Choose a log size based on how much logging activity occurs and the amount of risk in case of catastrophic failure. If you cannot afford to lose more than an hour's worth of data, create many small log files that each hold an hour's worth of transactions. Turn on continuous-log backup. Small logical-log files fill sooner, which means more frequent logical-log backups.

If your system is stable with high logging activity, choose larger logs to improve performance. Continuous-log backups occur less frequently with large log files. Also consider the maximum transaction rates and speed of the backup devices. Do not let the whole logical log fill. Turn on continuous-log backup and leave enough room in the logical logs to handle the longest transactions.

The backup process can hinder transaction processing that involves data located on the same disk as the logical-log files. If enough logical-log disk space is available, however, you can wait for periods of low user activity before you back up the logical-log files.

Estimating Logical-Log Size When Logging Dbspaces

You can use the following formula to obtain an initial estimate for LOGSIZE in kilobytes:

```
LOGSIZE = (connections * maxrows * rowsize) / 1024) / LOGFILES
```

connections is the maximum number of connections for all network types specified in the **sqlhosts** file or registry by one or more NETTYPE parameters. If you configured more than one connection by setting multiple NETTYPE configuration parameters in your configuration file, sum the **users** fields for each NETTYPE parameter, and substitute this total for *connections* in the preceding formula.

maxrows is the largest number of rows to be updated in a single transaction.

rowsize is the average size of a row in bytes. You can calculate *rowsize* by adding up the length (from the **syscolumns** system catalog table) of the columns in a row.

1024 is a necessary divisor because you specify LOGSIZE in kilobytes.

To obtain a better estimate, execute the **onstat -u** command during peak activity periods. The last line of the **onstat -u** output contains the maximum number of concurrent connections.

You need to adjust the size of the logical log when your transactions include simple large objects or smart large objects, as the following sections describe.

You also can increase the amount of space devoted to the logical log by adding another logical-log file, as your *Administrator's Guide* explains.

Estimating the Logical-Log Size When Logging Simple Large Objects

To obtain better overall performance for applications that perform frequent updates of TEXT or BYTE data in blobspaces, reduce the size of the logical log. Blobpages cannot be reused until the logical log to which they are allocated is backed up. When TEXT or BYTE data activity is high, the performance impact of more frequent checkpoints is balanced by the higher availability of free blobpages.

When you use volatile blobpages in blobspaces, smaller logs can improve access to simple large objects that must be reused. Simple large objects cannot be reused until the log in which they are allocated is flushed to disk. In this case, you can justify the cost in performance because those smaller log files are backed up more frequently.

IDS

Estimating Logical-Log Size When Logging Smart Large Objects

If you plan to log smart-large-object user data, you must ensure that the log size is considerably larger than the amount of data being written. Smart-large-object metadata is always logged even if the smart large objects are not logged.

Use the following guidelines when you log smart large objects:

- If you are appending data to a smart large object, the increased logging activity is roughly equal to the amount of data written to the smart large object.
- If you are updating a smart large object (overwriting data), the increased logging activity is roughly twice the amount of data written to the smart large object. The database server logs both the before-image and after-image of a smart large object for update transactions. When updating the smart large objects, the database server logs only the updated parts of the before and after image.
- Metadata updates affect logging less. Even though metadata is always logged, the number of bytes logged is usually much smaller than the smart large objects.

DYNAMIC_LOGS

The dynamic log file allocation feature prevents hangs caused by rollbacks of a long transaction because the database server does not run out of log space. Dynamic log allocation allows you to do the following actions:

- Add a logical log file while the system is active, even during fast recover.
- Insert a logical log file immediately after the current log file, instead of appending it to the end.
- Immediately access the logical log file even if the root dbspace is not backed up.

The default value for the DYNAMIC_LOGS configuration parameter is 2, which means that the database server automatically allocates a new logical log file after the current log file when it detects that the next log file contains an open transaction. The database server automatically checks if the log after the current log still contains an open transaction at the following times:

- Immediately after it switches to a new log file while writing log records (not while reading and applying log records)
- At the beginning of the transaction cleanup phase which occurs as the last phase of logical recovery

  Logical recovery happens at the end of fast recovery and at the end of a cold restore or roll forward. For more information on the phases of fast recovery, refer to your *Administrator's Guide*.
- During transaction cleanup (rollback of open transactions), a switch to a new log file log might occur

  The database server also checks after this switch because it is writing log records for the rollback.

When you use the default value of 2 for DYNAMIC_LOGS, the database server determines the location and size of the new logical log for you:

- The database server uses the following criteria to determine on which disk to allocate the new log file:
  - Favor mirrored dbspaces
  - Avoid root dbspace until no other critical dbspace is available
  - Least favored space is unmirrored and noncritical dbspaces
- The database server uses the average size of the largest log file and the smallest log file for the size of the new logical log file. If not enough contiguous disk space is available for this average size, the database server searches for space for the next smallest average size. The database server allocates a minimum of 200 kilobytes for the new log file.

If you want to control the location and size of the additional log file, set DYNAMIC_LOGS to 1. When the database server switches log files, it still checks if the next active log contains an open transaction. If it does find an open transaction in the next log to be active, it does the following actions:

- Issues alarm event 27 (log required)
- Writes a warning message to the online log
- Pauses to wait for the administrator to manually add a log with the **onparams -a -i** command-line option

You can write a script that will execute when alarm event 27 occurs to execute **onparams -a -i** with the location you want to use for the new log. Your script can also execute the **onstat -d** command to check for adequate space and execute the **onparams -a -i** command with the location that has enough space. You must use the **-i** option to add the new log right after the current log file.

If you set DYNAMIC_LOGS to 0, the database server still checks whether the next active log contains an open transaction when it switches log files. If it does find an open transaction in the next log to be active, it issues the following warning:

```
WARNING: The oldest logical log file (%d) contains records
from an open transaction (0x%p), but the Dynamic Log
Files feature is turned off.
```

LTXHWM and LTXEHWM

With the dynamic log file feature, the long transaction high watermarks are no longer as critical as in previous versions because the database server does not run out of log space unless you use up the physical disk space available on the system. Therefore, in Version 9.3, LTXHWM and LTXEHWM are not in the **onconfig.std** file and default to the following values, depending on the value of the DYNAMIC_LOGS configuration parameter:

- With dynamic log file allocation

  When you use the default value of 2 for DYNAMIC_LOGS or set DYNAMIC_LOGS to 1, the long transaction high watermark default values are:

  ```
  LTXHWM 80
  LTXEHWM 90
  ```

  The default values for LTXHWM and LTXEHWM are higher than in previous versions of the database server because the database server adds logical logs without the need to restart. Because the database server does not run out of logs, other users can still access the log during the rollback of a long transaction.

- Without dynamic log file allocation

  If you set DYNAMIC_LOGS to 0, the default values are 50 for LTXHWM and 60 for LTXEHWM.

The LTXHWM parameter still indicates how full the logical log is when the database server starts to check for a possible long transaction and to roll it back. LTXEHWM still indicates the point at which the database server suspends new transaction activity to locate and roll back a long transaction. These events should be rare, but if they occur, they can indicate a serious problem within an application.

Important: *Informix recommends that you keep these default values for LTXHWM and LTXEHWM.*

Under normal operations, use the default values for LTXHWM and LTXEHWM. However, you might want to change these default values for one of the following reasons:

- To allow other transactions to continue update activity (which requires access to the log) during the rollback of a long transaction

  In this case, you increase the value of LTXEHWM to raise the point at which the long transaction rollback has exclusive access to the log.
- To perform scheduled transactions of unknown length, such as large loads that are logged

  In this case, you increase the value of LTXHWM so that the transaction has a chance to complete before reaching the high watermark.

Configuration Parameters That Affect Page Cleaning

Another consideration is how page cleaning is performed. If pages are not cleaned often enough, an **sqlexec** thread that performs a query might be unable to find the available pages that it needs. It must then initiate a *foreground write* and wait for pages to be freed. Foreground writes impair performance, so you should avoid them. To reduce the frequency of foreground writes, increase the number of page cleaners or decrease the threshold for triggering a page cleaning. (See "LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY" on page 5-56.) Use **onstat -F** to monitor the frequency of foreground writes.

The following configuration parameters affect page cleaning:

- CLEANERS
- LRUS
- LRU_MAX_DIRTY
- LRU_MIN_DIRTY
- RA_PAGES
- RA_THRESHOLD

"RA_PAGES and RA_THRESHOLD" on page 5-40 describes the RA_PAGES and RA_THRESHOLD parameters.

CLEANERS

The CLEANERS configuration parameter indicates the number of page-cleaner threads to run. For installations that support fewer than 20 disks, Informix recommends one page-cleaner thread for each disk that contains database server data. For installations that support between 20 and 100 disks, Informix recommends one page-cleaner thread for every two disks. For larger installations, Informix recommends one page-cleaner thread for every four disks. If you increase the number of LRU queues as the previous section describes, increase the number of page-cleaner threads proportionally.

LRUS, LRU_MAX_DIRTY, and LRU_MIN_DIRTY

The LRUS configuration parameter indicates the number of least recently used (LRU) queues to set up within the shared-memory buffer pool. The buffer pool is distributed among LRU queues. Configuring more LRU queues allows more page cleaners to operate and reduces the size of each LRU queue. For a single-processor system, Informix suggests that you set the LRUS parameter to a minimum of 4. For multiprocessor systems, set the LRUS parameter to a minimum of 4 or NUMCPUVPS, whichever is greater.

Use LRUS with LRU_MAX_DIRTY and LRU_MIN_DIRTY to control how often pages are flushed to disk between full checkpoints.

To monitor the percentage of dirty pages in LRU queues, use the **onstat -R** command. When the number of dirty pages consistently exceeds the LRU_MAX_DIRTY limit, you have too few LRU queues or too few page cleaners. First, use the LRUS parameter to increase the number of LRU queues. If the percentage of dirty pages still exceeds LRU_MAX_DIRTY, use the CLEANERS parameter to increase the number of page cleaners.

Fuzzy Checkpoints and Page Cleaning

With fuzzy checkpoints, the buffer pool does not need to be cleaned as often as with full checkpoints. Therefore, you might increase transactional throughput if you increase the values of LRU_MAX_DIRTY and LRU_MIN_DIRTY. Retain the same gap between LRU_MAX_DIRTY and LRU_MIN_DIRTY.

To monitor the percentage of fuzzy pages in the buffer pool, use the **onstat -B** command. The checkpoint message in the online message log also tells you the number of pages not flushed to disk.

Configuration Parameters That Affect Backup and Restore

The following configuration parameters affect backup and restore:

- BAR_IDLE_TIMEOUT
- BAR_MAX_BACKUP
- BAR_NB_XPORT_COUNT
- BAR_PROGRESS_FREQ
- BAR_XFER_BUF_SIZE

UNIX

- LTAPEBLK
- LTAPEDEV
- LTAPESIZE
- TAPEBLK
- TAPEDEV
- TAPESIZE ♦

ON-Bar Configuration Parameters

The BAR_IDLE_TIMEOUT configuration parameter specifies the maximum number of minutes that an onbar-worker process is idle before it is shut down.

The BAR_MAX_BACKUP configuration parameter specifies the maximum number of backup processes per ON-Bar command. To control parallel execution of ON-Bar processes, use this configuration parameter.

BAR_NB_XPORT_COUNT specifies the number of shared-memory data buffers for each backup or restore process. BAR_PROGRESS_FREQ specifies in minutes how frequently the backup or restore progress messages display in the activity log. BAR_XFER_BUF_SIZE specifies the size in pages of the buffers.

For more information about these configuration parameters, refer to the *Backup and Restore Guide*.

UNIX

ontape Configuration Parameters

The LTAPEBLK, LTAPEDEV, and LTAPESIZE parameters specify the block size, device, and tape size for logical-log backups made with **ontape**.The TAPEBLK configuration parameter specifies the block size for database backups made with **ontape**, **onload**, and **onunload**. TAPEDEV specifies the tape device. TAPESIZE specifies the tape size for these backups. For information about these utilities and specific recommendations for backup-and-restore operations, refer to the *Backup and Restore Guide*.

Configuration Parameters That Affect Rollback and Recovery

The following configuration parameters affect fast recovery:

- OFF_RECVRY_THREADS
- ON_RECVRY_THREADS

The OFF_RECVRY_THREADS and ON_RECVRY_THREADS parameters specify the number of recovery threads, respectively, that operate when the database server performs a cold restore, warm restore, or fast recovery. The setting of OFF_RECVRY_THREADS controls cold restores, and the setting of ON_RECVRY_THREADS controls fast recovery and warm restores.

With fuzzy checkpoints, fast recovery might take longer than with full checkpoints. To improve the performance of fast recovery, increase the number of fast-recovery threads with the ON_RECVRY_THREADS parameter. The number of threads should usually match the number of tables or fragments that are frequently updated to roll forward the transactions recorded in the logical log.

Another estimate is the number of tables or fragments that experience frequent updates. On a single-CPU host, the number of threads should be no fewer than 10 and no more than 30 or 40. At a certain point, the overhead that is associated with each thread outweighs the advantages of parallel threads.

A warm restore takes place concurrently with other database operations. To reduce the impact of the warm restore on other users, you can allocate fewer threads to it than you would to a cold restore. However, to replay logical-log transactions in parallel during a warm restore, specify more threads in the ON_RECVRY_THREADS parameter.

Configuration Parameters That Affect Data Replication and Auditing

Data replication and auditing are optional. To obtain immediate performance improvements, you can disable these features, provided that the operating requirements for your system allow you to do so.

Data Replication

Data replication typically adds overhead to logical-log I/O that is proportional to your logging activity. A configuration that optimizes logging activity without data replication is likely to optimize logging with data replication.

The following configuration parameters affect data-replication performance:

- DRINTERVAL
- DRTIMEOUT

The DRINTERVAL parameter indicates whether the data-replication buffer is flushed synchronously or asynchronously to the secondary database server. If this parameter is set to flush asynchronously, it specifies the interval between flushes. Each flush impacts the CPU and sends data across the network to the secondary database server.

The DRTIMEOUT parameter specifies the interval for which either database server waits for a transfer acknowledgment from the other. If the primary database server does not receive the expected acknowledgment, it adds the transaction information to the file named in the DRLOSTFOUND configuration parameter. If the secondary database server receives no acknowledgment, it changes the data-replication mode as the DRAUTO configuration parameter specifies. For more information on data replication, refer to your *Administrator's Guide*.

Auditing

The effect of auditing on performance is largely determined by the auditing events that you choose to record. Depending on which users and events are audited, the impact of this feature can vary widely. Infrequent events, such as requests to connect to a database, have low performance impact. Frequent events, such as requests to read any row, can generate a large amount of auditing activity. The more users for whom such frequent events are audited, the greater the impact on performance. For information about auditing, refer to the *Trusted Facility Manual*.

The following configuration parameters affect auditing performance:

- ADTERR
- ADTMODE

The ADTERR parameter specifies whether the database server is to halt processing for a user session for which an audit record encounters an error. When ADTERR is set to halt such a session, the response time for that session appears to degrade until one of the successive attempts to write the audit record succeeds.

The ADTMODE parameter enables or disables auditing according to the audit records that you specify with the **onaudit** utility. Records are written to files in the directory that the AUDITPATH parameter specifies. The AUDITSIZE parameter specifies the size of each audit-record file.

Chapter

6

Table Performance Considerations

In This Chapter 6-5

Placing Tables on Disk. 6-5
- Isolating High-Use Tables 6-7
- Placing High-Use Tables on Middle Partitions of Disks 6-8
- Using Multiple Disks. 6-9
  - Using Multiple Disks for a Dbspace 6-9
  - Using Multiple Disks for Logical Logs 6-9
  - Spreading Temporary Tables and Sort Files Across Multiple Disks 6-10
- Backup-and-Restore Considerations 6-11
- Improving Performance for Nonfragmented Tables and Table Fragments 6-11

Estimating Table Size 6-12
- Estimating Data Pages 6-12
  - Estimating Tables with Fixed-Length Rows. 6-12
  - Estimating Tables with Variable-Length Rows. 6-15
  - Selecting an Intermediate Value for the Size of the Table . . . 6-16
- Estimating Pages That Simple Large Objects Occupy. 6-16
  - Storing Simple Large Objects in the Tblspace or a Separate Blobspace 6-18
  - Estimating Tblspace Pages for Simple Large Objects. 6-19

Managing Sbspaces. 6-20
- Estimating Pages That Smart Large Objects Occupy 6-20
  - Estimating the Size of the Sbspace and Metadata Area 6-20
  - Sizing the Metadata Area Manually for a New Chunk 6-21
- Improving Metadata I/O for Smart Large Objects 6-23

Monitoring Sbspaces 6-25
Using oncheck -cS 6-25
Using oncheck -pe 6-26
Using oncheck -pS 6-27
Using onstat -g smb 6-29
Changing Storage Characteristics of Smart Large Objects 6-30
Altering Smart-Large-Object Columns 6-33

Managing Extents 6-34
Choosing Table Extent Sizes 6-35
Extent Sizes for Tables in a Dbspace 6-35
Extent Sizes for Table Fragments 6-37
Extent Sizes for Smart Large Objects in Sbspaces 6-37
Monitoring Active Tblspaces 6-38
Managing Extents 6-39
Considering the Upper Limit on Extents 6-39
Checking for Extent Interleaving 6-41
Eliminating Interleaved Extents 6-42
Reclaiming Unused Space Within an Extent. 6-44

Changing Tables 6-46
Loading and Unloading Tables 6-46
Advantages of Logging Tables 6-47
Advantages of Nonlogging Tables 6-47
Dropping Indexes for Table-Update Efficiency 6-50
Attaching or Detaching Fragments 6-51
Altering a Table Definition 6-51
Slow Alter 6-51
In-Place Alter 6-52
Fast Alter 6-61

Denormalizing the Data Model to Improve Performance 6-61
Shortening Rows 6-62
Expelling Long Strings 6-62
Using VARCHAR Columns 6-62
Using TEXT Data 6-63
Moving Strings to a Companion Table 6-63
Building a Symbol Table 6-63

Splitting Wide Tables 6-64
Division by Bulk. 6-64
Division by Frequency of Use 6-64
Division by Frequency of Update 6-65
Costs of Companion Tables 6-65
Redundant Data 6-65
Adding Redundant Data 6-66

In This Chapter

This chapter describes performance considerations associated with unfragmented tables and table fragments. It covers the following issues:

- Table placement on disk to increase throughput and reduce contention
- Space estimates for tables and blobpages
- Sbspace management
- Extents management
- Changes to tables that add or delete historical data
- Denormalization of the database to reduce overhead

Placing Tables on Disk

Tables that the database server supports reside on one or more portions of a disk or disks. You control the placement of a table on disk when you create it by assigning it to a dbspace. A dbspace consists of one or more chunks. Each chunk corresponds to all or part of a disk partition. When you assign chunks to dbspaces, you make the disk space in those chunks available for storing tables or table fragments.

When you configure chunks and allocate them to dbspaces, you must relate the size of the dbspaces to the tables or fragments that each dbspace is to contain. To estimate the size of a table, follow the instructions in "Estimating Table Size" on page 6-12.

The database administrator (DBA) who is responsible for creating a table assigns that table to a dbspace in one of the following ways:

- By using the IN DBSPACE clause of the CREATE TABLE statement
- By using the dbspace of the current database

  The most recent DATABASE or CONNECT statement that the DBA issues before issuing the CREATE TABLE statement sets the current database.

The DBA can fragment a table across multiple dbspaces, as described in "Planning a Fragmentation Strategy" on page 9-3, or use the ALTER FRAGMENT statement to move a table to another dbspace. The ALTER FRAGMENT statement provides the simplest method for altering the placement of a table. However, the table is unavailable while the database server processes the alteration. Schedule the movement of a table or fragment at a time that affects the fewest users. For a description of the ALTER FRAGMENT statement, see the *Informix Guide to SQL: Syntax*.

Other methods exist for moving tables between dbspaces. A DBA can unload the data from a table and then move that data to another dbspace with the SQL statements LOAD and UNLOAD, as the *Informix Guide to SQL: Syntax* describes. The database server administrator can perform the same actions with the **onload** and **onunload** utilities, as the *Informix Migration Guide* describes, or with the High-Performance Loader (HPL), as the *Guide to the High-Performance Loader* describes.

Moving tables between databases with LOAD and UNLOAD, **onload** and **onunload**, or HPL involves periods in which data from the table is copied to tape and then reloaded onto the system. These periods present windows of vulnerability during which a table can become inconsistent with the rest of the database. To prevent the table from becoming inconsistent, you must restrict access to the version that remains on disk while the data transfers occur.

Depending on the size, fragmentation strategy, and indexes that are associated with a table, it can be faster to unload a table and reload it than to alter fragmentation. For other tables, it can be faster to alter fragmentation. You might have to experiment to determine which method is faster for a table that you want to move or repartition.

Isolating High-Use Tables

You can place a table with high I/O activity on a dedicated disk device and thus reduce contention for the data that is stored in that table. When disk drives have different performance levels, you can put the tables with the highest use on the fastest drives. Placing two high-use tables on separate disk devices reduces competition for disk access when the two tables experience frequent, simultaneous I/O from multiple applications or when joins are formed between them.

To isolate a high-use table on its own disk device, assign the device to a chunk, assign that chunk to a dbspace, and then place the table in the dbspace that you created. Figure 6-1 shows three high-use tables placed on three disks.

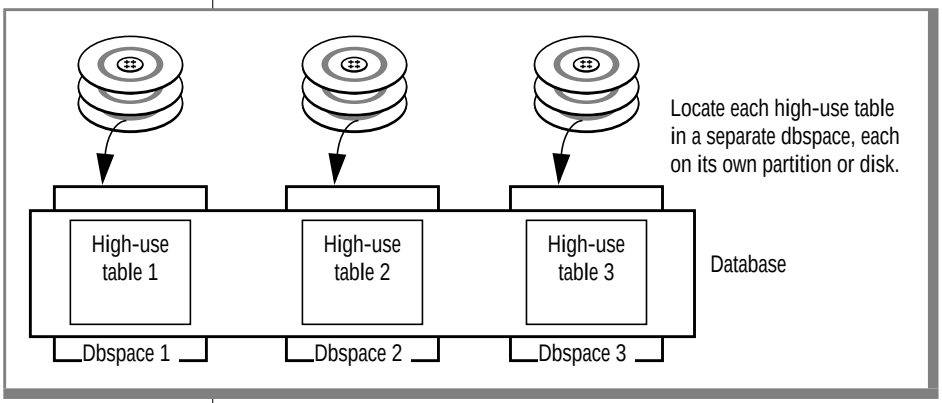


Figure 6-1
Isolating High-Use Tables

Placing High-Use Tables on Middle Partitions of Disks

To minimize disk-head movement, place the most frequently accessed data on partitions close to the middle of the disk, as Figure 6-2 shows. This approach minimizes disk-head movement to reach data in the high-demand table.

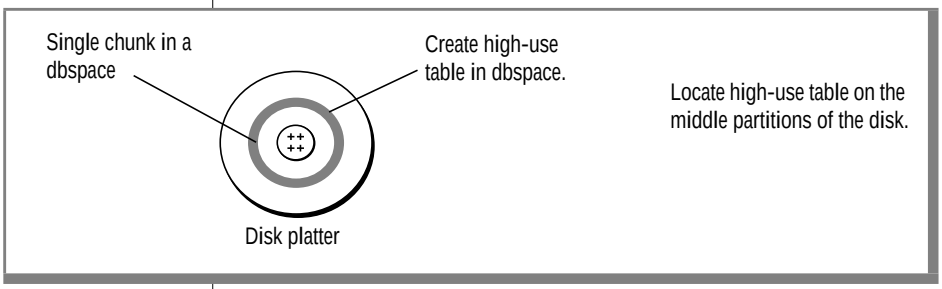

Single chunk in a dbspace
Create high-use table in dbspace.
Locate high-use table on the middle partitions of the disk.
Disk platter

Figure 6-2
Disk Platter with High-Use Table Located on Middle Partitions

To place high-use tables on the middle partition of the disk, create a raw device composed of cylinders that reside midway between the spindle and the outer edge of the disk. (For instructions on how to create a raw device, see the administrator's guide for your operating system.) Allocate a chunk, associating it with this raw device, as your *Administrator's Guide* describes. Then create a dbspace with this same chunk as the initial and only chunk. When you create a high-use table, place the table in this dbspace.

Using Multiple Disks

This section discusses using multiple disks for dbspaces, logical logs, and temporary dbspaces.

Using Multiple Disks for a Dbspace

A dbspace can include multiple chunks, and each chunk can represent a different disk. This arrangement allows you to distribute data in a dbspace over multiple disks. Figure 6-3 shows a dbspace distributed over multiple disks.

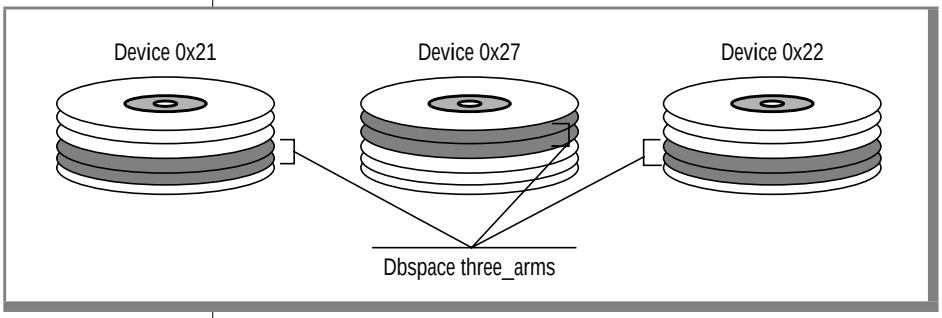


Figure 6-3
A Dbspace Distributed over Three Disks

Using multiple disks for a dbspace helps to distribute I/O across dbspaces that contain several small tables. Because you cannot use this type of distributed dbspace for parallel database queries (PDQ), Informix recommends that you use the table-fragmentation techniques described in "Designing a Distribution Scheme" on page 9-11 to partition large, high-use tables across multiple dbspaces.

Using Multiple Disks for Logical Logs

You can distribute logical logs in different dbspaces on multiple disks in round-robin fashion to improve logical backup performance. This scheme allows the database server to back up logs on one disk while logging on the other disks.

Informix recommends keeping your logical logs and the physical log on separate devices to improve performance by decreasing I/O contention on a single device. The logical and physical logs are created in the root dbspace when the database server is initialized. After initialization, you can move them to other dbspaces.

Spreading Temporary Tables and Sort Files Across Multiple Disks

To define several dbspaces for temporary tables and sort files, use **onspaces -t**. When you place these dbspaces on different disks and list them in the DBSPACETEMP configuration parameter, you can spread the I/O associated with temporary tables and sort files across multiple disks, as Figure 6-4 illustrates. You can list dbspaces that contain regular tables in DBSPACETEMP.

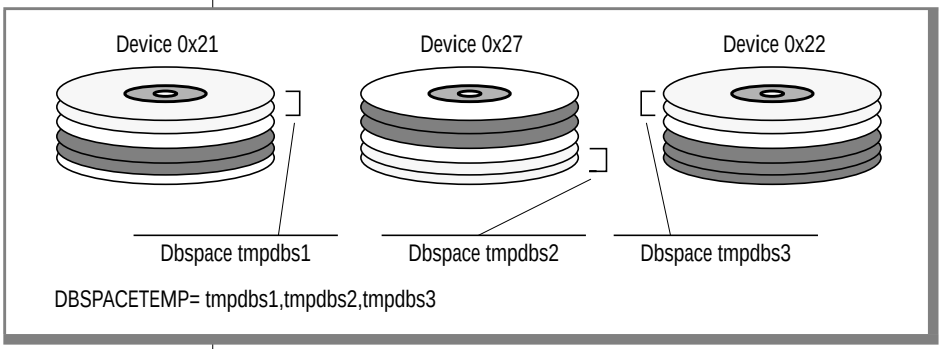


Figure 6-4
Dbspaces for Temporary Tables and Sort Files

Users can specify their own lists of dbspaces for temporary tables and sort files with the **DBSPACETEMP** environment variable. For details, refer to "Configuring Dbspaces for Temporary Tables and Sort Files" on page 5-12.

Backup-and-Restore Considerations

When you decide where to place your tables or fragments, remember that if a device that contains a dbspace fails, all tables or table fragments in that dbspace are rendered inaccessible, even though tables and fragments in other dbspaces are accessible. The need to limit data unavailability in the event of a disk failure might influence which tables you group together in a particular dbspace.

Although you must perform a cold restore if a dbspace that contains critical data fails, you need only perform a warm restore if a noncritical dbspace fails. The desire to minimize the impact of cold restores might influence the dbspace that you use to store critical data. For more information, see the *Archive and Backup Guide* or the *Backup and Restore Guide*.

Improving Performance for Nonfragmented Tables and Table Fragments

The following factors affect the performance of an individual table or table fragment:

- The placement of the table or fragment, as previous sections describe
- The size of the table or fragment
- The indexing strategy used
- The size and placement of table extents with respect to one another
- The frequency access rate to the table

Estimating Table Size

This section describes methods for calculating the approximate sizes (in disk pages) of tables.

For a description of size calculations for indexes, refer to "Estimating lndex Pages" on page 7-3.

The disk pages allocated to a table are collectively referred to as a *tblspace*. The tblspace includes data pages. A separate tblspace includes index pages. If simple large objects (TEXT or BYTE data) are associated with a table that is not stored in an alternative dbspace, pages that hold simple large objects are also included in the tblspace.

The tblspace does not correspond to any fixed region within a dbspace. The data extents and indexes that make up a table can be scattered throughout the dbspace.

The size of a table includes all the pages within the tblspace: data pages and pages that store simple large objects. Blobpages that are stored in a separate blobspace or on an optical subsystem are not included in the tblspace and are not counted as part of the table size. The following sections describe how to estimate the page count for each type of page within the tblspace.

Tip: *If an appropriate sample table already exists, or if you can build a sample table of realistic size with simulated data, you do not have to make estimates. You can run* ***oncheck -pt*** *to obtain exact numbers.*

Estimating Data Pages

How you estimate the data pages of a table depends on whether that table contains fixed-length or variable-length rows.

Estimating Tables with Fixed-Length Rows

Perform the following steps to estimate the size (in pages) of a table with fixed-length rows. A table with fixed-length rows has no columns of VARCHAR or NVARCHAR data type.

To estimate the page size, row size, number of rows, and number of data pages

1. Use **onstat -b** to obtain the size of a page.

   The **buffer size** field in the last line of this output displays the page size.

2. Subtract 28 from this amount to account for the header that appears on each data page.

   The resulting amount is referred to as *pageuse*.

3. To calculate the size of a row, add the widths of all the columns in the table definition. TEXT and BYTE columns each use 56 bytes.

   If you have already created your table, you can use the following SQL statement to obtain the size of a row:

   ```
   SELECT rowsize FROM systables WHERE tabname =
      'table-name';
   ```

4. Estimate the number of rows that the table is expected to contain.

   This number is referred to as *rows*.

   The procedure for calculating the number of data pages that a table requires differs depending on whether the row size is less than or greater than *pageuse*.

5. If the size of the row is less than or equal to *pageuse*, use the following formula to calculate the number of data pages.

   The **trunc()** function notation indicates that you are to round down to the nearest integer.

   ```
   data_pages = rows / trunc(pageuse/(rowsize + 4))
   ```

   The maximum number of rows per page is 255, regardless of the size of the row.

Important: *Although the maximum size of a row that the database server accepts is approximately 32 kilobytes, performance degrades when a row exceeds the size of a page. For information about breaking up wide tables for improved performance, refer to* *"Denormalizing the Data Model to Improve Performance" on page 6-61*.

6. If the size of the row is greater than *pageuse*, the database server divides the row between pages.

   The page that contains the initial portion of a row is called the *home page*. Pages that contains subsequent portions of a row are called *remainder pages*. If a row spans more than two pages, some of the remainder pages are completely filled with data from that row. When the trailing portion of a row uses less than a page, it can be combined with the trailing portions of other rows to fill out the partial remainder page. The number of data pages is the sum of the home pages, the full remainder pages, and the partial remainder pages.

   a. Calculate the number of home pages.

      The number of home pages is the same as the number of rows:

      ```
      homepages = rows
      ```

   b. Calculate the number of full remainder pages.

      First calculate the size of the row remainder with the following formula:

      ```
      remsize = rowsize - (pageuse + 8)
      ```

      If *remsize* is less than `pageuse - 4`, you have no full remainder pages.

      If *remsize* is greater than `pageuse - 4`, use *remsize* in the following formula to obtain the number of full remainder pages:

      ```
      fullrempages = rows * trunc(remsize/(pageuse - 8))
      ```

   c. Calculate the number of partial remainder pages.

      First calculate the size of a partial row remainder left after you have accounted for the home and full remainder pages for an individual row. In the following formula, the **remainder()** function notation indicates that you are to take the remainder after division:

      ```
      partremsize = remainder(rowsize/(pageuse - 8)) + 4
      ```

      The database server uses certain size thresholds with respect to the page size to determine how many partial remainder pages to use. Use the following formula to calculate the ratio of the partial remainder to the page:

      ```
      partratio = partremsize/pageuse
      ```

Use the appropriate formula in the following table to calculate the number of partial remainder pages.

partratio Value	Formula to Calculate the Number of Partial Remainder Pages
Less than .1	partrempages = *rows*/(trunc((*pageuse*/10)/*remsize*) + 1)
Less than .33	partrempages = *rows*/(trunc((*pageuse*/3)/*remsize*) + 1)
.33 or larger	partrempages = *rows*

d. Add up the total number of pages with the following formula:

```
tablesize = homepages + fullrempages + partrempages
```

Estimating Tables with Variable-Length Rows

When a table contains one or more VARCHAR or NVARCHAR columns, its rows can have varying lengths. These varying lengths introduce uncertainty into the calculations. You must form an estimate of the typical size of each VARCHAR column, based on your understanding of the data, and use that value when you make the estimates.

Important: *When the database server allocates space to rows of varying size, it considers a page to be full when no room exists for an additional row of the maximum size.*

To estimate the size of a table with variable-length rows, you must make the following estimates and choose a value between them, based on your understanding of the data:

- The maximum size of the table, which you calculate based on the maximum width allowed for all VARCHAR or NVARCHAR columns
- The projected size of the table, which you calculate based on a typical width for each VARCHAR or NVARCHAR column

To estimate the maximum number of data pages

1. To calculate *rowsize*, add together the maximum values for all column widths.
2. Use this value for *rowsize* and perform the calculations described in "Estimating Tables with Fixed-Length Rows" on page 6-12. The resulting value is called *maxsize*.

To estimate the projected number of data pages

1. To calculate *rowsize*, add together typical values for each of your variable-width columns. Informix suggests that you use the most frequently occurring width within a column as the typical width for that column. If you do not have access to the data or do not want to tabulate widths, you might choose to use some fractional portion of the maximum width, such as 2/3 (.67).
2. Use this value for *rowsize* and perform the calculations described in "Estimating Tables with Fixed-Length Rows" on page 6-12. The resulting value is called *projsize*.

Selecting an Intermediate Value for the Size of the Table

The actual table size should fall somewhere between *projsize* and *maxsize*. Based on your knowledge of the data, choose a value within that range that seems most reasonable to you. The less familiar you are with the data, the more conservative (higher) your estimate should be.

Estimating Pages That Simple Large Objects Occupy

The blobpages can reside in either the dbspace where the table resides or in a blobspace. For more information about when to use a blobspace, refer to "Storing Simple Large Objects in the Tblspace or a Separate Blobspace" on page 6-18.

The following methods for estimating blobpages yield a conservative (high) estimate because a single TEXT or BYTE column does not necessarily occupy the entire blobpage within a tblspace. In other words, a blobpage in a tblspace can contain multiple TEXT or BYTE columns.

To estimate the number of blobpages

1. Obtain the page size with **onstat -c**. Calculate the usable portion of the blobpage with the following formula:

   ```
   bpuse = pagesize - 32
   ```

2. For each byte of blobsize *n*, calculate the number of pages that the byte occupies (*bpages_n*) with the following formula:

   ```
   bpages1 = ceiling(bytesize1/bpuse)
   bpages2 = ceiling(bytesize2/bpuse)
   ...
   bpages_n = ceiling(bytesize_n/bpuse)
   ```

   The **ceiling()** function indicates that you should round up to the nearest integer value.

3. Add up the total number of pages for all simple large objects, as follows:

   ```
   blobpages = bpages1 + bpages2 + ... + bpagesn
   ```

Alternatively, you can base your estimate on the median size of simple large objects (TEXT or BYTE data); that is, the simple-large-object data size that occurs most frequently. This method is less precise, but it is easier to calculate.

To estimate the number of blobpages based on the median size of simple large objects

1. Calculate the number of pages required for simple large objects of median size, as follows:

   ```
   mpages = ceiling(mblobsize/bpuse)
   ```

2. Multiply this amount by the total number of simple large objects, as follows:

   ```
   blobpages = blobcount * mpages
   ```

Storing Simple Large Objects in the Tblspace or a Separate Blobspace

When you create a simple-large-object column on magnetic disk, you have the option of storing the column data in the tblspace or in a separate blobspace. You can often improve performance by storing simple-large-object data in a separate blobspace, as described in "Estimating Pages That Simple Large Objects Occupy" on page 6-16, and by storing smart large objects and user-defined data in sbspaces.

(You can also store simple large objects on optical media, but this discussion does not apply to simple large objects stored in this way.)

In the following example, a TEXT value is stored in the tblspace, and a BYTE value is stored in a blobspace named **rasters**:

```
CREATE TABLE examptab
   (
   pic_id SERIAL,
   pic_desc TEXT IN TABLE,
   pic_raster BYTE IN rasters
   )
```

A TEXT or BYTE value is always stored apart from the rows of the table; only a 56-byte descriptor is stored with the row. However, a simple large object occupies at least one disk page. The simple large object to which the descriptor points can reside in the same set of extents on disk as the table rows (in the same tblspace) or in a separate blobspace.

When simple large objects are stored in the tblspace, the pages of their data are interspersed among the pages that contain rows, which can greatly increase the size of the table. When the database server reads only the rows and not the simple large objects, the disk arm must move farther than when the blobpages are stored apart. The database server scans only the row pages in the following situations:

- When it performs any SELECT operation that does not retrieve a simple-large-object column
- When it uses a filter expression to test rows

Another consideration is that disk I/O to and from a dbspace is buffered in shared memory of the database server. Pages are stored in case they are needed again soon, and when pages are written, the requesting program can continue before the actual disk write takes place. However, because blobspace data is expected to be voluminous, disk I/O to and from blobspaces is not buffered, and the requesting program is not allowed to proceed until all output has been written to the blobspace.

For best performance, store a simple-large-object column in a blobspace in either of the following circumstances:

- When single data items are larger than one or two pages each
- When the number of pages of TEXT or BYTE data is more than half the number of pages of row data

Estimating Tblspace Pages for Simple Large Objects

In your estimate of the space required for a table, include blobpages for any simple large objects that are to be stored in that tblspace.

For a table that is both relatively small and nonvolatile, you can achieve the effect of a dedicated blobspace by separating row pages and blobpages, as the following steps explain.

To separate row pages from blobpages within a dbspace

1. Load the entire table with rows in which the simple-large-object columns are null.
2. Create all indexes.

   The row pages and the index pages are now contiguous.
3. Update all the rows to install the simple large objects.

   The blobpages now appear after the pages of row and index data within the tblspace.

Managing Sbspaces

This section describes the following topics concerning sbspaces:

- Estimating disk space
- Improving metadata I/O
- Monitoring
- Changing storage characteristics

Estimating Pages That Smart Large Objects Occupy

In your estimate of the space required for a table, you should also consider the amount of sbspace storage for any smart large objects (such as CLOB, BLOB, or multirepresentative data types) that are part of the table. An sbspace contains user-data areas and metadata areas. CLOB and BLOB data is stored in sbpages that reside in the user-data area. The metadata area contains the smart-large-object attributes, such as average size and whether or not the smart large object is logged. For more information about sbspaces, refer to your *Administrator's Guide*.

Estimating the Size of the Sbspace and Metadata Area

The first chunk of an sbspace must have a metadata area. When you add smart large objects, the database server adds more control information to this metadata area.

If you add a chunk to the sbspace after the initial allocation, you can take one of the following actions for metadata space:

- Allocate another metadata area on the new chunk by default.

  This action provides the following advantages:

  - It is easier because the database server automatically calculates and allocates a new metadata area on the added chunk based on the average smart large object size
  - Distributes I/O operations on the metadata area across multiple disks

- Use the existing metadata area

  If you specify the **onspaces -U** option, the database server does not allocate metadata space in the new chunk. Instead it must use a metadata area in one of the other chunks.

In addition, the database server reserves 40 percent of the user area to be used in case the metadata area runs out of space. Therefore, if the allocated metadata becomes full, the database server starts using this reserved space in the user area for additional control information.

You can let the database server calculate the size of the metadata area for you on the initial chunk and on each added chunks. However, you might want to specify the size of the metadata area explicitly, to ensure that the sbspace does not run out of metadata space and the 40 percent reserve area. You can use one of the following methods to explicitly specify the amount of metadata space to allocate:

- Specify the AVG_LO_SIZE tag on the **onspaces -Df** option.

  The database server uses this value to calculate the size of the metadata area to allocate when the **-Ms** option is not specified. If you do not specify AVG_LO_SIZE, the database server uses the default value of 8 kilobytes to calculate the size of the metadata area.

- Specify the metadata area size in the **-Ms** option of the **onspaces** utility.

  Use the procedure that "Sizing the Metadata Area Manually for a New Chunk" on page 6-21 describes to estimate a value to specify in the **onspaces -Ms** option.

For information on monitoring the space usage in an sbspace and allocating more space, refer to the managing storage chapter in your *Administrator's Guide*.

Sizing the Metadata Area Manually for a New Chunk

This procedure assumes that you know the sbspace size and need to allocate more metadata space. Each chunk can contain metadata but the sum total must accommodate enough room for all LO headers (average length 570 bytes each) and the chunk free list (which lists all the free extents in the chunk), as the following procedure shows.

To size the metadata area manually for a new chunk

1. Use the **onstat -d** option to obtain the size of the current metadata area from the **Metadata size** field.
2. Estimate the number of smart large objects that you expect to reside in the sbspace and their average size.
3. Use the following formula to calculate the total size of the metadata area:

```
Total metadata kilobytes = (LOcount*570)/1024 +
                           (numchunks*800) + 100
```

   LOcount is the number of smart large objects that you expect to have in all sbspace chunks, including the new one.

   numchunks is the total number of chunks in the sbspace.

4. To obtain the additional metadata area required, subtract the current metadata size that you obtained in step 1 from the value that you obtained in step 3.
5. When you add another chunk, specify in the **-Ms** option of the **onspaces -a** command the value that you obtained in step 4.

Example of Calculating the Metadata Area for a New Chunk

This example estimates the metadata size required for two sbspace chunks, using the preceding procedure:

1. Suppose the **Metadata size** field in the **onstat -d** option shows that the current metadata area is 1000 pages.

   If the system page size is 2048 bytes, the size of this metadata area is 2000 kilobytes, as the following calculation shows:

```
current metadata = (metadata_size * pagesize) / 1024
                 = (1000 * 2048) / 1024
                 = 2000 kilobytes
```

2. Suppose you expect 31,000 smart large objects in the two sbspace chunks

3. The following formula calculates the total size of metadata area required for both chunks, rounding up fractions:

```
Total metadata = (LOcount*570)/1024 + (numchunks*800) + 100
               = (31,000 * 570)/1024 + (2*800) + 100
               = 17256 + 1600 + 100
               = 18956 kilobytes
```

4. To obtain the additional metadata area required, subtract the current metadata size that you obtained in step 1 from the value that you obtained in step 3.

```
Additional metatdata = Total metatdata - current metadata
                     = 18956 - 2000
                     = 16956 kilobytes
```

5. When you add the chunk to the sbspace, use the **-Ms** option of the **onspaces -a** command to specify a metadata area of 16,956 kilobytes.

```
% onspaces -a sbchk2 -p /dev/raw_dev1 -o 200 -Ms 16956
```

For more information about **onspaces** and **onstat -d**, see the utilities chapter in the *Administrator's Reference*.

Improving Metadata I/O for Smart Large Objects

The metadata pages in an sbspace contain information about the location of the smart large objects in the sbspace. Typically, these pages are read intensive. You can distribute I/O to these pages in one of the following ways:

- Mirror the chunks that contain metadata.

  For more information about the implications of mirroring, refer to "Consider Mirroring for Critical Data Components" on page 5-8.

- Position the metadata pages on the fastest portion of the disk.

  Because the metadata pages are the most read-intensive part of an sbspace, place the metadata pages toward the middle of the disk to minimize disk seek time. To position metadata pages, use the **-Mo** option when you create the sbspace or add a chunk with the **onspaces** utility.

- Spread metadata pages across disks.

  To spread metadata pages across disks, create multiple chunks in an sbspace, with each chunk residing on a separate disk. When you add a chunk to the sbspace with the **onspaces** utility, specify the **-Ms** option to allocate pages for the metadata information.

  Although the database server attempts to keep the metadata information with its corresponding data in the same chunk, it cannot guarantee that they will be together.

- Decrease the number of extents each smart large object occupies.

  When a smart large object spans multiple extents, the metadata area contains a separate descriptor for each extent. To decrease the number of descriptor entries that must be read for each smart large object, specify the expected final size of the smart large object when you create the smart large object.

  The database server allocates the smart large object as a single extent (if it has contiguous storage in the chunk) when you specify the final size in either of the following functions:

  - The DataBlade API **mi_lo_specset_estbytes** function
  - The ESQL/C **ifx_lo_specset_estbytes** function

  For more information on the functions to open a smart large object and to set the estimated number of bytes, see the *Informix ESQL/C Programmer's Manual* and *DataBlade API Programmer's Manual*.

  For more information about sizing extents, refer to "Sbspace Extent Sizes" on page 5-30.

Important: *For highest data availability, mirror all sbspace chunks that contain metadata.*

Monitoring Sbspaces

For better I/O performance, the entire smart large objects should be allocated in one extent to be contiguous. For more information about sizing extents, refer to "Sbspace Extent Sizes" on page 5-30.

Contiguity provides the following I/O performance benefits:

- Minimizes the disk-arm motion
- Requires fewer I/O operations to read the smart large object
- When doing large sequential reads, can take advantage of light-weight I/O, which reads in larger blocks of data (60 kilobytes or more, depending on your platform) in a single I/O operation

You can use the following command-line utilities to monitor the effectiveness of I/O operations on smart large objects:

- **oncheck -cS, -pe** and **-pS**
- **onstat -g smb s** option

The following sections describe how to use these utility options to monitor sbspaces. For more information about **oncheck** and **onstat**, see the utilities chapter in the *Administrator's Reference*.

Using oncheck -cS

The **oncheck -cS** option checks smart-large-object extents and the sbspace partitions in the user-data area. Figure 6-5 shows an example of the output from the **-cS** option for **s9_sbspc**.

The values in the **Sbs#**, **Chk#**, and **Seq#** columns correspond to the **Space Chunk Page** value in the **-pS** output. The **Bytes** and **Pages** columns display the size of each smart large object in bytes and pages.

To calculate the average size of smart large objects, you can total the numbers in the **Size (Bytes)** column and then divide by the number of smart large objects. In Figure 6-5, the average number of bytes allocated is 2690, as the following calculation shows:

```
Average size in bytes = (15736 + 98 + 97 + 62 + 87 + 56) / 6
                      = 16136 / 6
                      = 2689.3
```

For information on how to specify smart large object sizes to influence extent sizes, refer to "Sbspace Extent Sizes" on page 5-30.

```
Validating space 's9_sbspc' ...

Large Objects
  ID                 Ref Size       Allocced       Creat   Last
  Sbs# Chk#  Seq#  Cnt    (Bytes)    Pages Extns Flags   Modified
  ---- ---- ----- ---- ---------- -------- ----- ----- ------------------------
     2    2     1    1      15736        8     1 N-N-H Thu Jun 25 16:59:12 1998
     2    2     2    1         98        1     1 N-K-H Thu Jun 25 16:59:12 1998
     2    2     3    1         97        1     1 N-K-H Thu Jun 25 16:59:12 1998
     2    2     4    1         62        1     1 N-K-H Thu Jun 25 16:59:12 1998
     2    2     5    1         87        1     1 N-K-H Thu Jun 25 16:59:12 1998
     2    2     6    1         56        1     1 N-K-H Thu Jun 25 16:59:12 1998
```

Figure 6-5
oncheck -cS Output

The **Extns** field shows the minimum extent size, in number of pages, allocated to each smart large object.

Using oncheck -pe

Execute **oncheck -pe** to display the following information to determine if the smart large objects occupy contiguous space within an sbspace:

- Identifies each smart large object with the term `SBLOBSpace LO`

  The three values in brackets following `SBLOBSpace LO` correspond to the **Sbs#**, **Chk#**, and **Seq#** columns in the **-cS** output.
- Offset of each smart large object
- Number of disk pages (*not* sbpages) used by each smart large object

Tip: *The **oncheck -pe** option provides information about sbspace use in terms of database server pages, not sbpages.*

Figure 6-6 shows sample output. In this example, the **size** field shows that the first smart large object occupies eight pages. Because the **offset** field shows that the first smart large object starts at page 53 and the second smart large object starts at page 61, the first smart large object occupies contiguous pages.

```
Chunk Pathname                                        Size      Used      Free
                                                      1000       940        60

    Description                                            Offset     Size
    ---------------------------------------------------- -------- --------
    RESERVED PAGES                                              0        2
    CHUNK FREELIST PAGE                                         2        1
    s9_sbspc:'informix'.TBLSpace                                3       50
    SBLOBSpace LO [2,2,1]                                      53        8
    SBLOBSpace LO [2,2,2]                                      61        1
    SBLOBSpace LO [2,2,3]                                      62        1
    SBLOBSpace LO [2,2,4]                                      63        1
    SBLOBSpace LO [2,2,5]                                      64        1
    SBLOBSpace LO [2,2,6]                                      65        1
...
```

Figure 6-6 *oncheck -pe Output That Shows Contiguous Space Use*

Using oncheck -pS

The **oncheck -pS** option displays information about smart-large-object extents and metadata areas in sbspace partitions. If you do not specify an sbspace name on the command line, **oncheck** checks and displays the metadata for all sbspaces. Figure 6-7 shows an example of the **-pS** output for **s9_sbspc**.

To display information about smart large objects, execute the following command:

```
oncheck -pS spacename
```

The **oncheck -pS** output displays the following information for each smart large object in the sbspace:

- Space chunk page
- Size in bytes of each smart large object
- Object ID that DataBlade API and ESQL/C functions use
- Storage characteristics of each smart large object

When you use **onspaces -c -S** to create an sbspace, you can use the **-Df** option to specify various storage characteristics for the smart large objects. You can use **onspaces -ch** to change attributes after the sbspace is created. The **Create Flags** field in the **oncheck -pS** output displays these storage characteristics and other attributes of each smart large object. In Figure 6-7, the **Create Flags** field shows LO_LOG because the LOGGING tag was set to ON in the **-Df** option.

```
Space Chunk Page = [2,2,2] Object ID = 987122917
  LO SW Version              4
  LO Object Version          1
  Created by Txid            7
  Flags               0x31 LO_LOG LO_NOKEEP_LASTACCESS_TIME LO_HIGH_INTEG
  Data Type                  0
  Extent Size                -1
  IO Size                    0
  Created                    Thu Apr 12 17:48:35 2001
  Last Time Modified         Thu Apr 12 17:48:43 2001
  Last Time Accessed         Thu Apr 12 17:48:43 2001
  Last Time Attributes Modified Thu Apr 12 17:48:43 2001
  Ref Count                  1
  Create Flags        0x31 LO_LOG LO_NOKEEP_LASTACCESS_TIME LO_HIGH_INTEG
  Status Flags               0x0 LO_FROM_SERVER
  Size (Bytes)               2048
  Size Limit                 -1
  Total Estimated Size       -1
  Deleting TxId              -1
  LO Map Size                200
  LO Map Last Row            -1
  LO Map Extents             2
  LO Map User Pages          2
```

Figure 6-7
oncheck -pS Output

Using onstat -g smb

Use the **onstat -g smb s** option to display the following characteristics that affect the I/O performance of each sbspace:

- Logging status

  If applications are updating temporary smart large objects, logging is not required. You can turn off logging to reduce the amount of I/O activity to the logical log, CPU utilization, and memory resources.

- Average smart-large-object size

  Average size and extent size should be similar to reduce the number of I/O operations required to read in an entire smart large object. The **avg s/kb** output field shows the average smart-large-object size in kilobytes. In Figure 6-8, the **avg s/kb** output field shows the value `30` kilobytes.

  Specify the final size of the smart large object in either of the following functions to allocate the object as a single extent:

  - The DataBlade API **mi_lo_specset_estbytes** function
  - The ESQL/C **ifx_lo_specset_estbytes** function

  For more information on the functions to open a smart large object and to set the estimated number of bytes, see the *Informix ESQL/C Programmer's Manual* and *DataBlade API Programmer's Manual*.

- First extent size, next extent size, and minimum extent size

  The **1st sz/p**, **nxt sz/p**, and **min sz/p** output fields show these extent sizes if you set the extent tags in the **-Df** option of **onspaces**. In Figure 6-8, these output fields show values of `0` and `-1` because these tags are not set in **onspaces**.

```
sbnum 7    address 2afae48
  Space    : flags    nchk owner    sbname
             -------- 1    informix client
  Defaults : LO_LOG LO_KEEP_LASTACCESS_TIME

  LO     : ud b/pg flags   flags    avg s/kb max lcks
           2048    0       -------- 30       -1
  Ext/IO : 1st sz/p nxt sz/p min sz/p mx io sz
           4        0        0        -1

  HdrCache : max   free
             512   0
```

Figure 6-8
onstat -g smb s Output

Changing Storage Characteristics of Smart Large Objects

When you create an sbspace but do not specify values in the **-Df** option of the **onspaces -c -S** command, you use the defaults for the storage characteristics and attributes (such as logging and buffering).

After you monitor sbspaces, you might want to change the storage characteristics, logging status, lock mode, or other attributes for new smart large objects.

The database administrator or programmer can use the following methods to override these default values for storage characteristics and attributes:

- The database administrator can use one of the following **onspaces** options:
  - Specify values when the sbspace is first created with the **onspaces -c -S** command.
  - Change values after the sbspace is created with the **onspaces -ch** command.

  Specify these values in the tag options of the **-Df** option of **onspaces**. For more information about the **onspaces** utility, refer to the utilities chapter in the *Administrator's Reference.*
- The database administrator can specify values in the PUT clause of the CREATE TABLE or ALTER TABLE statements.

  These values override the values in the **onspaces** utility and are valid only for smart large objects that are stored in the associated column of the specific table. Other smart large objects (from columns in other tables) might also reside in this same sbspace. These other columns continue to use the storage characteristics and attributes of the sbspace that **onspaces** defined (or the default values, if **onspaces** did not define them) unless these columns also used a PUT clause to override them for a particular column.

  If you do not specify the storage characteristics for a smart-large-object column in the PUT clause, they are inherited from the sbspace.

If you do not specify the PUT clause when you create a table with smart-large-object columns, the database server stores the smart large objects in the system default sbspace, which is specified by the SBSPACENAME configuration parameter in the ONCONFIG file. In this case, the storage characteristics and attributes are inherited from the SBSPACENAME sbspace.

DB API

E/C

- Programmers can use functions in the DataBlade API and ESQL/C to alter storage characteristics for a smart-large-object column.

  For information about the DataBlade API functions for smart large objects, refer to the *DataBlade API Programmer's Manual*. For information about the ESQL/C functions for smart large objects, see the *Informix ESQL/C Programmer's Manual*. ♦

Figure 6-9 summarizes the ways to alter the storage characteristics for a smart large object.

Figure 6-9
Altering Storage Characteristics and Other Attributes of an Sbspace

	System-Specified Storage Characteristics		Column-Level Storage Characteristics	Programmer-Specified Storage Characteristics	
Storage Characteristic or Attribute	**System Default Value**	**Specified by -Df Option in onspaces Utility**	**Specified by PUT clause of CREATE TABLE or ALTER TABLE**	**Specified by a DataBlade API Function**	**Specified by an ESQL/C Function**
Last-access time	OFF	ACCESSTIME	KEEP ACCESS TIME, NO KEEP ACCESS TIME	Yes	Yes
Lock mode	BLOB	LOCK_MODE	No	Yes	Yes
Logging status	OFF	LOGGING	LOG, NO LOG	Yes	Yes
Size of extent	None	EXTENT_SIZE	EXTENT SIZE	Yes	Yes
Size of next extent	None	NEXT_SIZE	No	No	No

(1 of 2)

Storage Characteristic or Attribute	System-Specified Storage Characteristics		Column-Level Storage Characteristics	Programmer-Specified Storage Characteristics	
	System Default Value	Specified by -Df Option in onspaces Utility	Specified by PUT clause of CREATE TABLE or ALTER TABLE	Specified by a DataBlade API Function	Specified by an ESQL/C Function
Minimum extent size	2 kilobytes on Windows NT 4 kilobytes on UNIX	MIN_EXT_SIZE	No	No	No
Size of smart large object	8 kilobytes	Average size of all smart large objects in sbspace: AVG_LO_SIZE	No	Estimated size of a particular smart large object Maximum size of a particular smart large object	Estimated size of a particular smart large object Maximum size of a particular smart large object
Buffer pool usage	ON	BUFFERING	No	LO_BUFFER and LO_NOBUFFER flags	LO_BUFFER and LO_NOBUFFER flags
Name of sbspace	SBSPACENAME	Not in **-Df** option. Name specified in **onspaces -S** *option*.	Name of an existing sbspace in which a smart large object resides: PUT ... IN clause	Yes	Yes
Fragmentation across multiple sbspaces	None	No	Round-robin distribution scheme: PUT ... IN clause	Round-robin or expression-based distribution scheme	Round-robin or expression-based distribution scheme

(2 of 2)

Altering Smart-Large-Object Columns

When you create a table, you have the following options for choosing storage characteristics and other attributes (such as logging status, buffering, and lock mode) for specific smart-large-object columns:

- Use the values that were set when the sbspace was created. These values are specified in one of the following ways:
  - With the various tags of the **-Df** option of the **onspaces -c -S** command
  - With the system default value for any specific tag that was not specified

  For guidelines to change the default storage characteristics of the **-Df** tags, refer to "onspaces Options That Affect Sbspace I/O" on page 5-29.
- Use the PUT clause of the CREATE TABLE statement to specify non-default values for particular characteristics or attributes.

  When you do not specify particular characteristics or attributes in the PUT clause, they default to the values set in the **onspaces -c -S** command.

Later, you can use the PUT clause of the ALTER TABLE statement to change the optional storage characteristics of these columns. Figure 6-9 on page 6-31 shows which characteristics and attributes you can change.

You can use the PUT clause of the ALTER TABLE statement to perform the following actions:

- Specify the smart-large-object characteristics and storage location when you add a new column to a table.

  The smart large objects in the new columns can have different characteristics than those in the existing columns.
- Change the smart-large-object characteristics of an existing column.

  The new characteristics of the column apply only to new smart large objects created for that column. The characteristics of existing smart large objects remain the same.

- Convert simple large objects to smart large objects by changing the column type from TEXT to CLOB or from BYTE to BLOB. For more information about converting simple large objects, see the *Informix Migration Guide*.

For example, the BLOB data in the **catalog** table in the **superstores_demo** database is stored in **s9_sbspc** with logging turned off and has an extent size of 100 kilobytes. You can use the PUT clause of the ALTER TABLE statement to turn on logging and store new smart large objects in a different sbspace. For information about sbspace logging, see smart large objects in the logging chapter of your *Administrator's Guide*.

For information about changing sbspace extents with the CREATE TABLE statement, refer to "Extent Sizes for Smart Large Objects in Sbspaces" on page 6-37.

For more information about CREATE TABLE and ALTER TABLE, see the *Informix Guide to SQL: Syntax*.

Managing Extents

As you add rows to a table, the database server allocates disk space in units called *extents*. Each extent is a block of physically contiguous pages from the dbspace. Even when the dbspace includes more than one chunk, each extent is allocated entirely within a single chunk, so that it remains contiguous.

Contiguity is important to performance. When the pages of data are contiguous, and when the database server reads the rows sequentially during read-ahead, light scans, or lightweight I/O operations, disk-arm motion is minimized. For more information on these operations, refer to "Sequential Scans" on page 5-37, "Light Scans" on page 5-38, and "Configuration Parameters That Affect Sbspace I/O" on page 5-28.

The mechanism of extents is a compromise between the following competing requirements:

- Most dbspaces are shared among several tables.
- The size of some tables is not known in advance.
- Tables can grow at different times and different rates.
- All the pages of a table should be adjacent for best performance.

Because table sizes are not known, the database server cannot preallocate table space. Therefore, the database server adds extents only as they are needed, but all the pages in any one extent are contiguous for better performance. In addition, when the database server creates a new extent that is adjacent to the previous one, it treats both as a single extent.

Choosing Table Extent Sizes

When you create a table, you can specify extent sizes for the following storage spaces:

- Data rows of a table in a dbspace
- Each fragment of a fragmented table
- Smart large objects in an sbspace

Extent Sizes for Tables in a Dbspace

When you create a table, you can specify the size of the first extent as well as the size of the extents to be added as the table grows. The following example creates a table with a 512-kilobyte initial extent and 100-kilobyte added extents:

```
CREATE TABLE big_one (…column specifications…)
   IN big_space
   EXTENT SIZE 512
   NEXT SIZE 100
```

The default value for the extent size and the next-extent size is eight times the disk page size on your system. For example, if you have a 2-kilobyte page, the default length is 16 kilobytes.

To change the size of extents to be added, use the ALTER TABLE statement. This change does not affect extents that already exist. The following example changes the next-extent size of the table to 50 kilobytes:

```
ALTER TABLE big_one MODIFY NEXT SIZE 50
```

The next-extent sizes of the following kinds of tables do not affect performance significantly:

- A small table is defined as a table that has only one extent. If such a table is heavily used, large parts of it remain buffered in memory.
- An infrequently used table is not important to performance no matter what size it is.
- A table that resides in a dedicated dbspace always receives new extents that are adjacent to its old extents. The size of these extents is not important because, being adjacent, they perform as one large extent.

When you assign an extent size to these kinds of tables, the only consideration is to avoid creating large numbers of extents. A large number of extents causes the database server to spend extra time finding the data. In addition, an upper limit exists on the number of extents allowed. ("Considering the Upper Limit on Extents" on page 6-39 covers this topic.)

No upper limit exists on extent sizes except the size of the chunk. When you know the final size of a table (or can confidently predict it within 25 percent), allocate all its space in the initial extent. When tables grow steadily to unknown size, assign them next-extent sizes that let them share the dbspace with a small number of extents each. The following steps outline one possible approach.

To allocate space for table extents

1. Decide how to allocate space among the tables.

   For example, you might divide the dbspace among three tables in the ratio 0.4: 0.2: 0.3 (reserving 10 percent for small tables and overhead).
2. Give each table one-fourth of its share of the dbspace as its initial extent.
3. Assign each table one-eighth of its share as its next-extent size.
4. Monitor the growth of the tables regularly with **oncheck**.

As the dbspace fills up, you might not have enough contiguous space to create an extent of the specified size. In this case, the database server allocates the largest contiguous extent that it can.

Extent Sizes for Table Fragments

When you fragment an existing table, you might want to adjust the next-extent size because each fragment requires less space than the original, unfragmented table. If the unfragmented table was defined with a large next-extent size, the database server uses that same size for the next-extent on *each* fragment, which results in over-allocation of disk space. Each fragment requires only a proportion of the space for the entire table.

For example, if you fragment the preceding **big_one** sample table across five disks, you can alter the next-extent size to one-fifth the original size. For more information about the ALTER FRAGMENT statement, see the *Informix Guide to SQL: Syntax*. The following example changes the next-extent size to one-fifth of the original size:

```
ALTER TABLE big_one MODIFY NEXT SIZE 20
```

Extent Sizes for Smart Large Objects in Sbspaces

When you create a table, Informix recommends that you use one of the following extent sizes for smart large objects in the sbspace:

- Extent size that the database server calculates for you
- Final size of the smart large object, as indicated by one of the following functions when you open the sbspace in an application program:

  DB API

  - The DataBlade API **mi_lo_specset_estbytes** function

    For more information on the DataBlade API functions to open a smart large object and set the estimated number of bytes, see the *DataBlade API Programmer's Manual.* ♦

  E/C

  - The ESQL/C **ifx_lo_specset_estbytes** function

    For more information on the ESQL/C functions to open a smart large object and set the estimated number of bytes, see the *Informix ESQL/C Programmer's Manual.* ♦

For more information about sizing extents, refer to "Sbspace Extent Sizes" on page 5-30. For more information, refer to "Monitoring Sbspaces" on page 6-25.

Monitoring Active Tblspaces

Monitor tblspaces to determine which tables are active. Active tables are those that a thread has currently opened.

Output from the **onstat -t** option includes the tblspace number and the following four fields.

Field	Description
npages	Pages allocated to the tblspace
nused	Pages used from this allocated pool
nextns	Number of extents used
npdata	Number of data pages used

If a specific operation needs more pages than are available (**npages** minus **nused**), a new extent is required. If enough space is available in this chunk, the database server allocates the extent here; if not, the database server looks for space in other available chunks. If none of the chunks contains adequate contiguous space, the database server uses the largest block of contiguous space that it can find in the dbspace. Figure 6-10 shows an example of the output from this option.

```
Tblspaces
 n address  flgs ucnt tblnum   physaddr npages nused  npdata nrows  nextns
 0 422528   1    1    100001   10000e   150    124    0      0      3
 1 422640   1    1    200001   200004   50     36     0      0      1
54 426038   1    6    100035   1008ac   3650   3631   3158   60000  3
62 4268f8   1    6    100034   1008ab   8      6      4      60     1
63 426a10   3    6    100036   1008ad   368    365    19     612    3
64 426b28   1    6    100033   1008aa   8      3      1      6      1
193 42f840  1    6    10001b   100028   8      5      2      30     1
 7 active, 200 total, 64 hash buckets
```

Figure 6-10
onstat -t Output

Managing Extents

This section covers the following topics:

- Considering upper limit on number of extents
- Checking for extent interleaving
- Eliminating extent interleaving
- Reclaiming unused space within an extent

Considering the Upper Limit on Extents

Do not allow a table to acquire a large number of extents because an upper limit exists on the number of extents allowed. Trying to add an extent after you reach the limit causes error -136 (`No more extents`) to follow an INSERT request.

To determine the upper limit on number of extents allowed for a table

1. Run the following **oncheck** option to obtain the physical address of the object (table or index fragment) for which you wish to calculate extent limits.

   ```
   oncheck -pt databasename:tablename
   ```

   Figure 6-11 shows sample output for **oncheck -pt**.

```
TBLspace Report for stores7:wbyrne.sfe_enquiry
Physical Address 7002c7
Number of special columns 18
Number of keys 0
Number of extents 65
Number of data pages 960
```

Figure 6-11
oncheck -pt Output

2. Split the physical address of the table or index fragment into the chunk# (leading digits) and page# (last 5 digits), and then run **oncheck -pP chunk# page#**, specifying the arguments as hexadecimal numbers by prefixing with 0x.

   In Figure 6-11, the **Physical Address** is 7002c7. Therefore, **chunk#** is 0x007 (or 0x07) and the **page#** is 0x0002c (or 0x2c) in the following **oncheck** command:

   ```
   oncheck -pP 0x7 0x2c
   ```

   Figure 6-12 shows sample output for o**ncheck -pp 0x7 0x2c**.

```
addr   stamp  nslots flag type  frptr frcnt next prev
7002c7 112686 5      2    PARTN 828   1196  0    0
slot ptr len flg
1     24  92   0
2    116  40   0
3    156 144   0
4    300   0   0
5    300 528   0
```

Figure 6-12
oncheck -pp chunk# page# Output

3. From the output of the **oncheck -pP**, take the number below the **frcnt** column and divide it by 8 to obtain the number of additional extents you can have for this object.

   In the sample **oncheck -pP** in Figure 6-12, the **frcnt** column shows the value 1196. The following calculation shows the number of additional:

   ```
   Additional_extents = trunc (frcnt / 8)
                      = trunc (1196 / 8)
                      = 149
   ```

4. To obtain the maximum number of extents, add the value in the **Number of extents** line in the **oncheck -pt** output to the Additional_extents value, as the following formula shows:

   ```
   Maximum_number_extents = Additional_extents +
      Number_of_extents
   ```

   In the sample **oncheck -pt** in Figure 6-11, the **Number of extents** line shows the value 65. The following calculation shows the maximum-number of extents for this table:

   ```
   Maximum_number_extents = 149 + 65 = 214
   ```

To help ensure that the limit is not exceeded, the database server performs the following actions:

- The database server checks the number of extents each time that it creates a new extent. If the number of the extent being created is a multiple of 16, the database server automatically doubles the next-extent size for the table. Therefore, at every sixteenth creation, the database server doubles the next-extent size.
- When the database server creates a new extent adjacent to the previous extent, it treats both extents as a single extent.

Checking for Extent Interleaving

When two or more growing tables share a dbspace, extents from one tblspace can be placed between extents from another tblspace. When this situation occurs, the extents are said to be *interleaved*. Interleaving creates gaps between the extents of a table, as Figure 6-13 shows. Performance suffers when disk seeks for a table must span more than one extent, particularly for sequential scans.

Try to optimize the table-extent sizes to allocate contiguous disk space, which limits head movement. Also consider placing the tables in separate dbspaces.

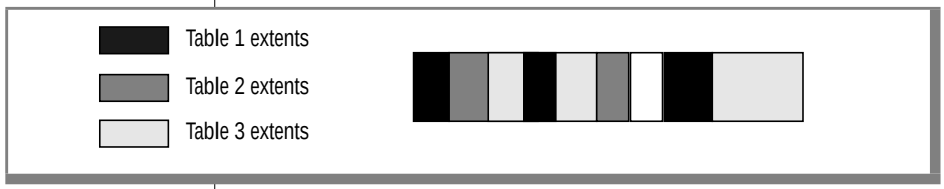


Figure 6-13
Interleaved Table Extents

Check periodically for extent interleaving by monitoring chunks. Execute **oncheck -pe** to obtain the physical layout of information in the chunk. The following information appears:

- Dbspace name and owner
- Number of chunks in the dbspace
- Sequential layout of tables and free space in each chunk
- Number of pages dedicated to each table extent or free space

This output is useful for determining the degree of extent interleaving. If the database server cannot allocate an extent in a chunk despite an adequate number of free pages, the chunk might be badly interleaved.

Eliminating Interleaved Extents

You can eliminate interleaved extents with one of the following methods:

- Reorganize the tables with the UNLOAD and LOAD statements.
- Create or alter an index to cluster.
- Use the ALTER TABLE statement.

Reorganizing Dbspaces and Tables to Eliminate Extent Interleaving

You can rebuild a dbspace to eliminate interleaved extents, as Figure 6-14 illustrates. The order of the reorganized tables within the dbspace is not important, but the pages of each reorganized table should be contiguous so that no lengthy seeks are required to read the table sequentially. When the disk arm reads a table nonsequentially, it ranges only over the space that table occupies.

Figure 6-14
A Dbspace Reorganized to Eliminate Interleaved Extents

To reorganize tables in a dbspace

D/B

1. Copy the tables in the dbspace individually to tape with the UNLOAD statement in DB-Access. ♦
2. Drop all the tables in the dbspace.
3. Re-create the tables with the LOAD statement or the **dbload** utility.

The LOAD statement re-creates the tables with the same properties they had before, including the same extent sizes.

You can also unload a table with the **onunload** utility and reload the table with the companion **onload** utility. For further information about selecting the correct utility or statement to use, refer to the *Informix Migration Guide*.

Creating or Altering an Index to Cluster

Depending on the circumstances, you can eliminate extent interleaving if you create a clustered index or alter an index to cluster. When you use the TO CLUSTER clause of the CREATE INDEX or ALTER INDEX statement, the database server sorts and reconstructs the table. The TO CLUSTER clause reorders rows in the physical table to match the order in the index. For more information, refer to "Clustering" on page 7-15.

The TO CLUSTER clause eliminates interleaved extents under the following conditions:

- The chunk must contain contiguous space that is large enough to rebuild each table.
- The database server must use this contiguous space to rebuild the table.

  If blocks of free space exist before this larger contiguous space, the database server might allocate the smaller blocks first. The database server allocates space for the ALTER INDEX process from the beginning of the chunk, looking for blocks of free space that are greater than or equal to the size that is specified for the next extent. When the database server rebuilds the table with the smaller blocks of free space that are scattered throughout the chunk, it does not eliminate extent interleaving.

To display the location and size of the blocks of free space, execute the **oncheck -pe** command.

To use the TO CLUSTER clause of the ALTER INDEX statement

1. For each table in the chunk, drop all fragmented or detached indexes except the one that you want to cluster.
2. Cluster the remaining index with the TO CLUSTER clause of the ALTER INDEX statement.

   This step eliminates interleaving the extents when you rebuild the table by rearranging the rows.
3. Re-create all the other indexes.

   You compact the indexes in this step because the database server sorts the index values before it adds them to the B-tree.

You do not need to drop an index before you cluster it. However, the ALTER INDEX process is faster than CREATE INDEX because the database server reads the data rows in cluster order using the index. In addition, the resulting indexes are more compact.

To prevent the problem from recurring, consider increasing the size of the tblspace extents. For more information, see the *Informix Guide to SQL: Tutorial*.

Using ALTER TABLE to Eliminate Extent Interleaving

If you use the ALTER TABLE statement to add or drop a column or to change the data type of a column, the database server copies and reconstructs the table. When the database server reconstructs the entire table, it rewrites the table to other areas of the dbspace. However, if other tables are in the dbspace, no guarantee exists that the new extents will be adjacent to each other.

Important: *For certain types of operations that you specify in the ADD, DROP, and MODIFY clauses, the database server does not copy and reconstruct the table during the ALTER TABLE operation. In these cases, the database server uses an in-place alter algorithm to modify each row when it is updated (rather than during the ALTER TABLE operation). For more information about the conditions for this in-place alter algorithm, refer to* *"In-Place Alter" on page 6-52**.*

Reclaiming Unused Space Within an Extent

Once the database server allocates disk space to a tblspace as part of an extent, that space remains dedicated to the tblspace. Even if all extent pages become empty after you delete data, the disk space remains unavailable for use by other tables.

Important: *When you delete rows in a table, the database server reuses that space to insert new rows into the same table. This section describes procedures to reclaim unused space for use by other tables.*

You might want to resize a table that does not require the entire amount of space that was originally allocated to it. You can reallocate a smaller dbspace and release the unneeded space for other tables to use.

As the database server administrator, you can reclaim the disk space in empty extents and make it available to other users by rebuilding the table. To rebuild the table, use any of the following SQL statements:

- ALTER INDEX
- UNLOAD and LOAD
- ALTER FRAGMENT

Reclaiming Space in an Empty Extent with ALTER INDEX

If the table with the empty extents includes an index, you can execute the ALTER INDEX statement with the TO CLUSTER clause. Clustering an index rebuilds the table in a different location within the dbspace. All the extents associated with the previous version of the table are released. Also, the newly built version of the table has no empty extents.

For more information about the syntax of the ALTER INDEX statement, refer to the *Informix Guide to SQL: Syntax*. For more information about clustering, refer to "Clustering" on page 7-15.

Reclaiming Space in an Empty Extent with the UNLOAD and LOAD Statements or the onunload and onload Utilities

If the table does not include an index, you can unload the table, re-create the table (either in the same dbspace or in another one), and reload the data with the UNLOAD and LOAD statements or the **onunload** and **onload** utilities.

For further information about selecting the correct utility or statement, refer to the *Informix Migration Guide*. For more information about the syntax of the UNLOAD and LOAD statements, refer to the *Informix Guide to SQL: Syntax*.

Releasing Space in an Empty Extent with ALTER FRAGMENT

You can use the ALTER FRAGMENT statement to rebuild a table, which releases space within the extents that were allocated to that table. For more information about the syntax of the ALTER FRAGMENT statement, refer to the *Informix Guide to SQL: Syntax*.

Changing Tables

You might want to change an existing table for various reasons:

- To refresh large decision-support tables with data periodically
- To add or drop historical data from a certain time period
- To add, drop, or modify columns in large decision-support tables when the need arises for different data analysis

Loading and Unloading Tables

Databases for decision-support applications are often created by periodically loading tables that have been unloaded from active OLTP databases. You can use one or more of the following methods to load large tables quickly:

- High-Performance Loader (HPL)

  You can use HPL in express mode to load tables quickly. For more information on how the database server performs high-performance loading, refer to the *Guide to the High-Performance Loader*.

- Nonlogging Tables

  The database server provides support to:

  - Create nonlogging or logging tables in a logging database.
  - Alter a table from nonlogging to logging and vice versa.

  The two table types are STANDARD (logging tables) and RAW (nonlogging tables). You can use any loading utility such as **dbimport** or HPL to load raw tables.

The following sections describe:

- Advantages of logging and nonlogging tables
- Step-by-step procedures to load data using nonlogging tables

For recovery information about standard and raw tables, refer to your *Administrator's Guide*.

Advantages of Logging Tables

The STANDARD type, which corresponds to a table in a logged database of previous versions, is the default. When you issue the CREATE TABLE statement without specifying the table type, you create a standard table.

Standard tables have the following features:

- Logging to allow rollbacks and fast recovery
- Recovery from backups
- All insert, delete, and update operations
- Constraints to maintain the integrity of your data
- Indexes to quickly retrieve a small number of rows

OLTP applications usually use standard tables. OLTP applications typically have the following characteristics:

- Real time insert, update, and delete transactions

  Logging and recovery of these transactions is critical to preserve the data. Locking is critical to allow concurrent access and to ensure the consistency of the data selected.

- Update, insert, or delete one row or a few rows at a time

  Indexes speed access to these rows. An index requires only a few I/O operations to access the pertinent row, but scanning a table to find the pertinent row might require many I/O operations.

Advantages of Nonlogging Tables

The advantage of nonlogging tables is that you can load very large data warehousing tables quickly because they have following characteristics:

- They do not use CPU and I/O resources for logging.
- They avoid problems such as running out of logical-log space.
- They are locked exclusively during an express load so that no other user can access the table during the load.
- Raw tables do not support indexes, referential constraints, and unique constraints, so overhead for constraint-checking and index-building is eliminated.

To quickly load a large, existing standard table

1. Drop indexes, referential constraints, and unique constraints.
2. Change the table to nonlogging.

   The following sample SQL statement changes a STANDARD table to nonlogging:

   ```
   ALTER TABLE largetab TYPE(RAW);
   ```
3. Load the table using a load utility such as **dbexport** or the High-Performance Loader (HPL).

   For more information on **dbexport** and **dbload**, refer to the *Informix Migration Guide*. For more information on HPL, refer to the *Guide to the High-Performance Loader*.
4. Perform a level-0 backup of the nonlogging table.

   You must make a level-0 backup of any nonlogging table that has been modified before you convert it to STANDARD type. The level-0 backup provides a starting point from which to restore the data.
5. Change the nonlogging table to a logging table before you use it in a transaction.

   The following sample SQL statement changes a raw table to a standard table:

   ```
   ALTER TABLE largetab TYPE(STANDARD);
   ```

Warning: *Informix recommends that you not use nonlogging tables within a transaction where multiple users can modify the data. If you need to use a nonlogging table within a transaction, either set Repeatable Read isolation level or lock the table in exclusive mode to prevent concurrency problems.*

   For more information on standard tables, see the previous section, "Advantages of Logging Tables."
6. Re-create indexes, referential constraints, and unique constraints.

To quickly load a new, large table

1. Create a nonlogging table in a logged database.

   The following sample SQL statements creates a nonlogging table:

   ```
   CREATE DATABASE history WITH LOG;
   CONNECT TO DATABASE history;
   CREATE RAW TABLE history (...
   );
   ```

2. Load the table using a load utility such as **dbexport** or the High-Performance Loader (HPL).

   For more information on **dbexport** and **dbload**, refer to the *Informix Migration Guide*. For more information on HPL, refer to the *Guide to the High-Performance Loader*.

3. Perform a level-0 backup of the nonlogging table.

   You must make a level-0 backup of any nonlogging table that has been modified before you convert it to STANDARD type. The level-0 backup provides a starting point from which to restore the data.

4. Change the nonlogging table to a logging table before you use it in a transaction.

   The following sample SQL statement changes a raw table to a standard table:

   ```
   ALTER TABLE largetab TYPE(STANDARD);
   ```

Warning: *Informix recommends that you not use nonlogging tables within a transaction where multiple users can modify the data. If you need to use a nonlogging table within a transaction, either set Repeatable Read isolation level or lock the table in exclusive mode to prevent concurrency problems.*

   For more information on standard tables, see the previous section, "Advantages of Logging Tables."

5. Create indexes on columns most often used in query filters.
6. Create any referential constraints and unique constraints, if needed.

Dropping Indexes for Table-Update Efficiency

In some applications, you can confine most table updates to a single time period. You can set up your system so that all updates are applied overnight or on specified dates.

When updates are performed as a batch, you can drop all nonunique indexes while you make updates and then create new indexes afterward. This strategy can have two positive effects:

- The updating program runs much faster if it does not have to update indexes at the same time that it updates tables.
- Re-created indexes are more efficient.

For more information about when to drop indexes, see "Dropping Indexes" on page 7-17.

To load a table that has no indexes

1. Drop the table (if it exists).
2. Create the table without specifying any unique constraints.
3. Load all rows into the table.
4. Alter the table to apply the unique constraints.
5. Create the nonunique indexes.

If you cannot guarantee that the loaded data satisfies all unique constraints, you must create unique indexes before you load the rows. You save time if the rows are presented in the correct sequence for at least one of the indexes. If you have a choice, make it the row with the largest key. This strategy minimizes the number of leaf pages that must be read and written.

Attaching or Detaching Fragments

Many customers use ALTER FRAGMENT ATTACH and DETACH statements to perform data warehouse-type operations. ALTER FRAGMENT DETACH provides a way to delete a segment of the table data rapidly. Similarly, ALTER FRAGMENT ATTACH provides a way to load large amounts of data into an existing table incrementally by taking advantage of the fragmentation technology.

For more information about how to take advantage of the performance enhancements for the ATTACH and DETACH options of the ALTER FRAGMENT statement, refer to "Improving the Performance of Attaching and Detaching Fragments" on page 9-29.

Altering a Table Definition

The database server uses one of several algorithms to process an ALTER TABLE statement in SQL:

- Slow alter
- In-place alter
- Fast alter

Slow Alter

When the database server uses the slow alter algorithm to process an ALTER TABLE statement, the table can be unavailable to other users for a long period of time because the database server:

- Locks the table in exclusive mode for the duration of the ALTER TABLE operation
- Makes a copy of the table in order to convert the table to the new definition
- Converts the data rows during the ALTER TABLE operation
- Can treat the ALTER TABLE statement as a long transaction and abort it if the LTXHWM threshold is exceeded

The database server uses the slow alter algorithm when the ALTER TABLE statement makes column changes that it cannot perform in place:

- Adding or drop a column created with the ROWIDS keyword
- Dropping a column of the TEXT or BYTE data type
- Modifying the data type of a column so that some possible values of the old data type cannot be converted to the new data type

  For example, if you modify a column of data type INTEGER to CHAR(n), the database server uses the slow alter algorithm if the value of n is less than 11. An INTEGER requires 10 characters plus one for the minus sign for the lowest possible negative values.
- Modifying the data type of a fragmentation column in a way that value conversion might cause rows to move to another fragment
- Adding, dropping or modifying any column when the table contains user-defined data types or smart large objects.

In-Place Alter

The in-place alter algorithm provides the following performance advantages over the slow alter algorithm:

- Increases table availability

  Other users can access the table sooner when the ALTER TABLE operation uses the in-place alter algorithm, because the database server locks the table for only the time that it takes to update the table definition and rebuild indexes that contain altered columns.

  This increase in table availability can increase system throughput for application systems that require 24 by 7 operation.

When the database server uses the in-place alter algorithm, it locks the table for a shorter time than the slow alter algorithm because the database server:

- Does not make a copy of the table to convert the table to the new definition
- Does not convert the data rows during the ALTER TABLE operation
- Alters the physical columns in place with the latest definition after the alter operation when you subsequently update or insert rows. The database server converts the rows that reside on each page that you updated.

- Requires less space than the slow alter algorithm

  When the ALTER TABLE operation uses the slow alter algorithm, the database server makes a copy of the table to convert the table to the new definition. The ALTER TABLE operation requires space at least twice the size of the original table plus log space.

  When the ALTER TABLE operation uses the in-place alter algorithm, the space savings can be substantial for very large tables.

- Improves system throughput during the ALTER TABLE operation

  The database server does not need to log any changes to the table data during the in-place alter operation. Not logging changes has the following advantages:

  - Log space savings can be substantial for very large tables.
  - The alter operation is not a long transaction.

When the Database Server Uses the In-Place Alter Algorithm

The database server uses the in-place alter algorithm for certain types of operations that you specify in the ADD, DROP, and MODIFY clauses of the ALTER TABLE statement:

- Add a column or list of columns of any data type except columns that you add with the ROWIDS keyword.
- Drop a column of any data type except TEXT or BYTE and columns created with the ROWIDS keyword.
- Add or drop a column created with the CRCOLS keyword.

- Modify a column for which the database server can convert all possible values of the old data type to the new data type.
- Modify a column that is part of the fragmentation expression if value changes do not require a row to move from one fragment to another fragment after conversion.

Important: *When a table contains a user-defined data type or smart large objects, the database server does not use the in-place alter algorithm even when the column being altered contains a built-in data type.*

Figure 6-15 shows the conditions under which the ALTER TABLE MODIFY statement uses the in-place alter algorithm to process the ALTER TABLE MODIFY Statement.

Figure 6-15
MODIFY Operations and Conditions That Use the In-Place Alter Algorithm

Operation on Column	Condition
Convert a SMALLINT column to a INTEGER column	All
Convert a SMALLINT column to an INTEGER8 column	All
Convert a SMALLINT column to a DEC(p2,s2) column	p2-s2 >= 5
Convert a SMALLINT column to a DEC(p2) column	p2-s2 >= 5 OR nf
Convert a SMALLINT column to a SMALLFLOAT column	All
Convert a SMALLINT column to a FLOAT column	All
Convert a SMALLINT column to a CHAR(n) column	n >= 6 AND nf
Convert a SMALLINT column to a SERIAL column	All
Convert a SMALLINT column to a SERIAL8 column	All
Convert an INT column to an INTEGER8 column	All
Convert an INT column to a DEC(p2,s2) column	p2-s2 >= 10
Convert an INT column to a DEC(p2) column	p2 >= 10 OR nf
Convert an INT column to a SMALLFLOAT column	nf
Convert an INT column to a FLOAT column	All

(1 of 3)

Operation on Column	Condition
Convert an INT column to a CHAR(n) column	n >= 11 AND nf
Convert an INT column to a SERIAL column	All
Convert an INT column to a SERIAL8 column	All
Convert a SERIAL column to an INTEGER8 column	All
Convert an SERIAL column to a DEC(p2,s2) column	p2-s2 >= 10
Convert a SERIAL column to a DEC(p2) column	p2 >= 10 OR nf
Convert a SERIAL column to a SMALLFLOAT column	nf
Convert a SERIAL column to a FLOAT column	All
Convert a SERIAL column to a CHAR(n) column	n >= 11 AND nf
Convert a SERIAL column to a SERIAL column	All
Convert a SERIAL column to a SERIAL8 column	All
Convert a DEC(p1,s1) column to a SMALLINT column	p1-s1 < 5 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to an INTEGER column	p1-s1 < 10 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to an INTEGER8 column	p1-s1 < 20 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to a SERIAL column	p1-s1 < 10 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to a SERIAL8 column	p1-s1 < 20 AND (s1 == 0 OR nf)
Convert a DEC(p1,s1) column to a DEC(p2,s2) column	p2-s2 >= p1-s1 AND (s2 >= s1 OR nf)
Convert a DEC(p1,s1) column to a DEC(p2) column	p2 >= p1 OR nf
Convert a DEC(p1,s1) column to a SMALLFLOAT column	nf
Convert a DEC(p1,s1) column to a FLOAT column	nf

(2 of 3)

Operation on Column	Condition
Convert a DEC(p1,s1) column to a CHAR(n) column	n >= 8 AND nf
Convert a DEC(p1) column to a DEC(p2) column	p2 >= p1 OR nf
Convert a DEC(p1) column to a SMALLFLOAT column	nf
Convert a DEC(p1) column to a FLOAT column	nf
Convert a DEC(p1) column to a CHAR(n) column	n >= 8 AND nf
Convert a SMALLFLOAT column to a DEC(p2) column	nf
Convert a SMALLFLOAT column to a FLOAT column	nf
Convert a SMALLFLOAT column to a CHAR(n) column	n >= 8 AND nf
Convert a FLOAT column to a DEC(p2) column	nf
Convert a FLOAT column to a SMALLFLOAT column	nf
Convert a FLOAT column to a CHAR(n) column	n >= 8 AND nf
Convert a CHAR(m) column to a CHAR(n) column	n >= m OR (nf AND not ANSI mode)
Increase the length of a CHARACTER column	Not ANSI mode
Increase the length of a DECIMAL or MONEY column	All

Notes:

- The column type DEC(p) refers to non-ANSI databases in which this data type is handled as floating point.
- In ANSI databases, DEC(p) defaults to DEC(p,0) and uses the same alter algorithm as DEC(p,s).
- The condition `nf` indicates that the database server uses the in-place alter algorithm when the modified column is not part of the table fragmentation expression.
- The condition `All` indicates that the database server uses the in-place alter algorithm for all cases of the specific column operation.

(3 of 3)

Performance Considerations for DML Statements

Each time you execute an ALTER TABLE statement that uses the in-place alter algorithm, the database server creates a new version of the table structure. The database server keeps track of all versions of table definitions. The database server resets the version status and all of the version structures and alter structures until the entire table is converted to the final format or a slow alter is performed.

If the database server detects any down-level version page during the execution of data manipulation language (DML) statements (INSERT, UPDATE, DELETE, SELECT), it performs the following actions:

- For UPDATE statements, the database server converts the entire data page or pages to the final format.
- For INSERT statements, the database server converts the inserted row to the final format and inserts it in the best-fit page. The database server converts the existing rows on the best-fit page to the final format.
- For DELETE statements, the database server does not convert the data pages to the final format.
- For SELECT statements, the database server does not convert the data pages to the final format.

  If your query accesses rows that are not yet converted to the new table definition, you might notice a slight degradation in the performance of your individual query, because the database server reformats each row before it is returned.

Performance Considerations for DDL Statements

The **oncheck -pT** *tablename* option displays data-page versions for outstanding in-place alter operations. An in-place alter is outstanding when data pages still exist with the old definition.

Figure 6-16 displays a portion of the output that the following **oncheck** command produces after four in-place alter operations are executed on the **customer** demonstration table:

```
oncheck -pT stores_demo:customer
```

```
...
Home Data Page Version Summary

            Version  Count

            0 (oldest)  2
            1        0
            2        0
            3        0
            4 (current)  0
...
```

Figure 6-16
Sample oncheck -pT Output for customer Table

The **Count** field in Figure 6-16 displays the number of pages that currently use that version of the table definition. This **oncheck** output shows that four versions are outstanding:

- A value of `2` in the **Count** field for the oldest version indicates that two pages use the oldest version.
- A value of `0` in the **Count** fields for the next four versions indicates that no pages have been converted to the latest table definition.

Important: *As you perform more in-place alters on a table, each subsequent ALTER TABLE statement takes more time to execute than the previous statement. Therefore, Informix recommends that you have no more than approximately 50 to 60 outstanding alters on a table. A large number of outstanding alters affects only the subsequent ALTER TABLE statements, but does not degrade the performance of SELECT statements.*

You can convert data pages to the latest definition with a dummy UPDATE statement. For example, the following statement, which sets a column value to the existing value, causes the database server to convert data pages to the latest definition:

```
UPDATE tab1 SET col1 = col1;
```

This statement does not change any data values, but it converts the format of the data pages to the latest definition.

After an update is executed on all pages of the table, the **oncheck -pT** command displays the total number of data pages in the **Count** field for the current version of the table.

Alter Operations That Do Not Use the In-Place Alter Algorithm

The database server does not use the in-place alter algorithm in the following situations:

- When more than one algorithm is in use

  If the ALTER TABLE statement contains more than one change, the database server uses the algorithm with the lowest performance in the execution of the statement.

  For example, assume that an ALTER TABLE MODIFY statement converts a SMALLINT column to a DEC(8,2) column and converts an INTEGER column to a CHAR(8) column. The conversion of the first column is an in-place alter operation, but the conversion of the second column is a slow alter operation. The database server uses the slow alter algorithm to execute this statement.

- When values have to move to another fragment

  For example, suppose you have a table with two integer columns and the following fragment expression:

  ```
  col1 < col2 in dbspace1, remainder in dbspace2
  ```

  If you execute the ALTER TABLE MODIFY statement in the following section, the database server stores a row `(4, 30)` in **dbspace1** before the alter but stores it in **dbspace2** after the alter operation because `4 < 30` but `"30" < "4"`.

Altering a Column That Is Part of an Index

If the altered column is part of an index, the table is still altered in place, but in this case the database server rebuilds the index or indexes implicitly. If you do not need to rebuild the index, you should drop or disable it before you perform the alter operation. Taking these steps improves performance.

However, if the column that you modify is a primary key or foreign key and you want to keep this constraint, you must specify those keywords again in the ALTER TABLE statement, and the database server rebuilds the index.

For example, suppose you create tables and alter the parent table with the following SQL statements:

```
CREATE TABLE parent
   (si smallint primary key constraint pkey);
CREATE TABLE child
   (si smallint references parent on delete cascade
   constraint ckey);
INSERT INTO parent (si) VALUES (1);
INSERT INTO parent (si) VALUES (2);
INSERT INTO child (si) VALUES (1);
INSERT INTO child (si) VALUES (2);
ALTER TABLE parent
    MODIFY (si int PRIMARY KEY constraint PKEY);
```

This ALTER TABLE example converts a SMALLINT column to an INT column. The database server retains the primary key because the ALTER TABLE statement specifies the PRIMARY KEY keywords and the PKEY constraint. However, the database server drops any referential constraints to that primary key. Therefore, you must also specify the following ALTER TABLE statement for the child table:

```
ALTER TABLE child
   MODIFY (si int references parent on delete cascade
         constraint ckey);
```

Even though the ALTER TABLE operation on a primary key or foreign key column rebuilds the index, the database server still takes advantage of the in-place alter algorithm. The in-place alter algorithm provides the following performance benefits:

- It does not make a copy of the table in order to convert the table to the new definition.
- It does not convert the data rows during the alter operation.
- It does not rebuild all indexes on the table.

Warning: *If you alter a table that is part of a view, you must re-create the view to obtain the latest definition of the table.*

Fast Alter

The database server uses the fast alter algorithm when the ALTER TABLE statement changes attributes of the table but does not affect the data. The database server uses the fast alter algorithm when you use the ALTER TABLE statement to:

- Change the next-extent size.
- Add or drop a constraint.
- Change the lock mode of the table.
- Change the unique index attribute without modifying the column type.

With the fast alter algorithm, the database server holds the lock on the table for just a short time. In some cases, the database server locks the system catalog tables only to change the attribute. In either case, the table is unavailable for queries for only a short time.

Denormalizing the Data Model to Improve Performance

The entity-relationship data model that the *Informix Guide to SQL: Tutorial* describes produces tables that contain no redundant or derived data. According to the tenets of relational database theory, these tables are well structured.

Sometimes, to meet extraordinary demands for high performance, you might have to modify the data model in ways that are undesirable from a theoretical standpoint. This section describes some modifications and their associated costs.

Shortening Rows

Usually, tables with shorter rows yield better performance than those with longer rows because disk I/O is performed in pages, not in rows. The shorter the rows of a table, the more rows occur on a page. The more rows per page, the fewer I/O operations it takes to read the table sequentially, and the more likely it is that a nonsequential access can be performed from a buffer.

The entity-relationship data model puts all the attributes of one entity into a single table for that entity. For some entities, this strategy can produce rows of awkward lengths. To shorten the rows, you can break columns into separate tables that are associated by duplicate key values in each table. As the rows get shorter, query performance should improve.

Expelling Long Strings

The most bulky attributes are often character strings. To make the rows shorter, you can remove them from the entity table. You can use the following methods to expel long strings:

- Use VARCHAR columns.
- Use TEXT data.
- Move strings to a companion table.
- Build a symbol table.

Using VARCHAR Columns

GLS

A database might contain CHAR columns that you can convert to VARCHAR columns. You can use a VARCHAR column to shorten the average row length when the average length of the text string in the CHAR column is at least 2 bytes shorter than the width of the column. For information about other character data types, refer to the *Informix Guide to GLS Functionality*. ®

VARCHAR data is immediately compatible with most existing programs, forms, and reports. You might need to recompile any forms produced by application development tools to recognize VARCHAR columns. Always test forms and reports on a sample database after you modify the table schema.

Using TEXT Data

When a string fills half a disk page or more, consider converting it to a TEXT column in a separate blobspace. The column within the row page is only 56 bytes long, which allows more rows on a page than when you include a long string. However, the TEXT data type is not automatically compatible with existing programs. The application needed to fetch a TEXT value is a bit more complicated than the code for fetching a CHAR value into a program.

Moving Strings to a Companion Table

Strings that are less than half a page waste disk space if you treat them as TEXT data, but you can move them from the main table to a companion table.

Building a Symbol Table

If a column contains strings that are not unique in each row, you can move those strings to a table in which only unique copies are stored.

For example, the **customer.city** column contains city names. Some city names are repeated in the column, and most rows have some trailing blanks in the field. Using the VARCHAR data type eliminates the blanks but not the duplication.

You can create a table named **cities**, as the following example shows:

```
CREATE TABLE cities (
   city_num SERIAL PRIMARY KEY,
   city_name VARCHAR(40) UNIQUE
)
```

You can change the definition of the **customer** table so that its **city** column becomes a foreign key that references the **city_num** column in the **cities** table.

To insert the city of the new customer into **cities**, you must change any program that inserts a new row into **customer.** The database server return code in the **SQLCODE** field of the SQL Communications Area (SQLCA) can indicate that the insert failed because of a duplicate key. It is not a logical error; it simply means that an existing customer is located in that city. For more information about the SQLCA, refer to the *Informix Guide to SQL: Tutorial*.

Besides changing programs that insert data, you must also change all programs and stored queries that retrieve the city name. The programs and stored queries must use a join to the new **cities** table in order to obtain their data. The extra complexity in programs that insert rows and the extra complexity in some queries is the result of giving up theoretical correctness in the data model. Before you make the change, be sure that it returns a reasonable savings in disk space or execution time.

Splitting Wide Tables

Consider all the attributes of an entity that has rows that are too wide for good performance. Look for some theme or principle to divide them into two groups. Split the table into two tables, a primary table and a companion table, repeating the primary key in each one. The shorter rows allow you to query or update each table quickly.

Division by Bulk

One principle on which you can divide an entity table is bulk. Move the bulky attributes, which are usually character strings, to the companion table. Keep the numeric and other small attributes in the primary table. In the demonstration database, you can split the **ship_instruct** column from the **orders** table. You can call the companion table **orders_ship**. It has two columns, a primary key that is a copy of **orders.order_num** and the original **ship_instruct** column.

Division by Frequency of Use

Another principle for division of an entity is frequency of use. If a few attributes are rarely queried, move them to a companion table. In the demonstration database, for example, perhaps only one program queries the **ship_instruct**, **ship_weight**, and **ship_charge** columns. In that case, you can move them to a companion table.

Division by Frequency of Update

Updates take longer than queries, and updating programs lock index pages and rows of data during the update process, preventing querying programs from accessing the tables. If you can separate one table into two companion tables, one with the most-updated entities and the other with the most-queried entities, you can often improve overall response time.

Costs of Companion Tables

Splitting a table consumes extra disk space and adds complexity. Two copies of the primary key occur for each row, one copy in each table. Two primary-key indexes also exist. You can use the methods described in earlier sections to estimate the number of added pages.

You must modify existing programs, reports, and forms that use SELECT * because fewer columns are returned. Programs, reports, and forms that use attributes from both tables must perform a join to bring the tables together.

In this case, when you insert or delete a row, two tables are altered instead of one. If you do not coordinate the alteration of the two tables (by making them within a single transaction, for example), you lose semantic integrity.

Redundant Data

Normalized tables contain no redundant data. Every attribute appears in only one table. Normalized tables also contain no derived data. Instead, data that can be computed from existing attributes is selected as an expression based on those attributes.

Normalizing tables minimizes the amount of disk space used and makes updating the tables as easy as possible. However, normalized tables can force you to use joins and aggregate functions often, and those processes can be time consuming.

As an alternative, you can introduce new columns that contain redundant data, provided you understand the trade-offs involved.

Adding Redundant Data

A correct data model avoids redundancy by keeping any attribute only in the table for the entity that it describes. If the attribute data is needed in a different context, you join tables to make the connection. But joining takes time. If a frequently used join affects performance, you can eliminate it by duplicating the joined data in another table.

In the **stores_demo** database, the **manufact** table contains the names of manufacturers and their delivery times. An actual working database might contain many other attributes of a supplier, such as address and sales representative name.

The contents of **manufact** are primarily a supplement to the **stock** table. Suppose that a time-critical application frequently refers to the delivery lead time of a particular product but to no other column of **manufact**. For each such reference, the database server must read two or three pages of data to perform the lookup.

You can add a new column, **lead_time**, to the **stock** table and fill it with copies of the **lead_time** column from the corresponding rows of **manufact**. That arrangement eliminates the lookup and therefore speeds up the application.

Like derived data, redundant data takes space and poses an integrity risk. In the example described in the previous paragraph, many extra copies of the lead time for each manufacturer can exist. (Each manufacturer can appear in **stock** many times.) The programs that insert or update a row of **manufact** must also update multiple rows of **stock**.

The integrity risk is simply that the redundant copies of the data might not be accurate. If a lead time is changed in **manufact**, the **stock** column is outdated until it is also updated. As you do with derived data, define the conditions under which redundant data might be wrong.

For more information on database design, refer to the *Informix Guide to Database Design and Implementation*.

Chapter

7

Index Performance Considerations

In This Chapter 7-3

Estimating lndex Pages 7-3
- Index Extent Sizes 7-4
  - Estimating Extent Size of Attached Index 7-4
  - Estimating Extent Size of Detached Index 7-4
- Estimating Conventional lndex Pages 7-5
- Estimating Index Pages for Spatial and User-Defined Data 7-10
  - B-Tree Indexes 7-10
  - R-Tree Indexes 7-10
  - Indexes That DataBlade Modules Provide 7-11

Managing Indexes 7-11
- Space Costs of Indexes 7-11
- Time Costs of Indexes 7-12
- Choosing Columns for Indexes 7-13
  - Filtered Columns in Large Tables 7-13
  - Order-By and Group-By Columns 7-14
  - Avoiding Columns with Duplicate Keys 7-14
  - Clustering 7-15
- Dropping Indexes 7-17

Improving Performance for Index Builds 7-18
- Estimating Sort Memory 7-19
- Estimating Temporary Space for Index Builds 7-20

Improving Performance for Index Checks 7-21

Indexes on User-Defined Data Types 7-22
Defining Indexes for User-Defined Data Types 7-23
B-Tree Secondary-Access Method 7-24
Determining the Available Access Methods 7-25
Using a User-Defined Secondary-Access Method 7-27
Using a Functional Index 7-28
Using an Index That a DataBlade Module Provides 7-31
Choosing Operator Classes for Indexes. 7-31
Operator Classes 7-31
Built-In B-Tree Operator Class 7-33
Determining the Available Operator Classes 7-35
Using an Operator Class 7-36

In This Chapter

This chapter describes performance considerations associated with indexes. It discusses space considerations, choosing indexes to create, and managing indexes for Dynamic Server.

Estimating Index Pages

The index pages associated with a table can add significantly to the size of a dbspace. By default, the database server creates the index in the same dbspace as the table, but in a separate tblspace from the table. To place the index in a separate dbspace, specify the IN keyword in the CREATE INDEX statement. For information about dbspaces and what objects are stored in the tblspace, see the chapter on data storage in your *Administrator's Guide*.

Although you cannot explicitly specify the extent size of an index, you can estimate the number of pages that an index might occupy to determine if your dbspace or dbspaces have enough space allocated.

Index Extent Sizes

The database server determines the extent size of an index based on the extent size for the corresponding table, regardless of whether the index is fragmented or not fragmented.

Estimating Extent Size of Attached Index

For an attached index, the database server uses the ratio of the index key size to the row size to assign an appropriate extent size for the index, as the following formula shows:

```
Index extent size = (index_key_size / table_row_size) *
   table_extent_size
```

index_key_size is the total widths of the indexed column or columns.

table_row_size is the sum of all the columns in the row.

table_extent_size is the value that you specify in the EXTENT SIZE keyword of the CREATE TABLE statement.

The database server also uses this same ratio for the next-extent size for the index:

```
Index next extent size = (index_key_size/table_row_size)*
   table_next_extent_size
```

Estimating Extent Size of Detached Index

For a detached index, the database server uses the ratio of the index key size plus some overhead bytes to the row size to assign an appropriate extent size for the index, as the following formula shows:

```
Detached Index extent size = ( (index_key_size + 9) /
                               table_row_size) * table_extent_size
```

For example, suppose you have the following values:

```
index_key_size = 8 bytes
table_row_size = 33 bytes
table_extent_size = 150 * 2-kilobyte page
```

The above formula calculates the extent size as follows:

```
Detached Index extent size = ( (8 + 9) / 33) * 150 * 2-kilobyte page
                           = (17/33) * 300 kilobytes
                           = 154 kilobytes
```

Estimating Conventional Index Pages

As Figure 7-1 shows, an index is arranged as a hierarchy of pages (technically, a *B-tree*). The topmost level of the hierarchy contains a single *root page*. Intermediate levels, when needed, contain *branch pages*. Each branch page contains entries that refer to a subset of pages in the next level of the index. The bottom level of the index contains a set of *leaf pages*. Each leaf page contains a list of index entries that refer to rows in the table.

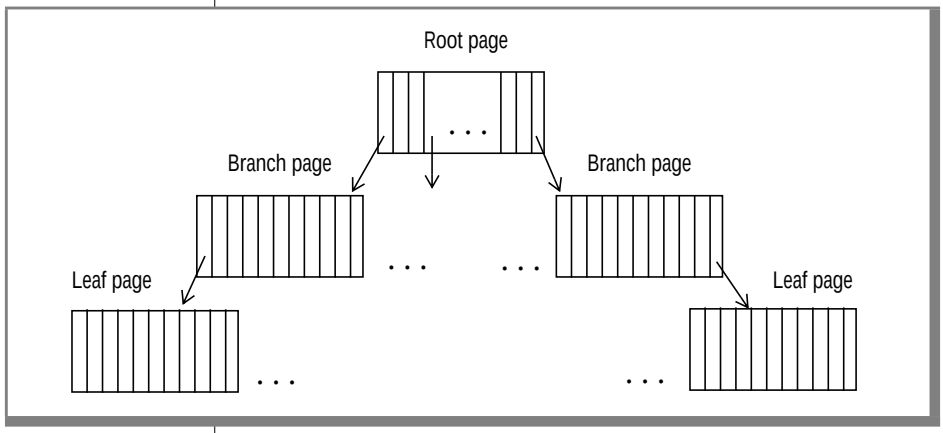


Figure 7-1
B-Tree Structure of an Index

The number of levels needed to hold an index depends on the number of unique keys in the index and the number of index entries that each page can hold. The number of entries per page depends, in turn, on the size of the columns being indexed.

If the index page for a given table can hold 100 keys, a table of up to 100 rows requires a single index level: the root page. When this table grows beyond 100 rows, to a size between 101 and 10,000 rows, it requires a two-level index: a root page and between 2 and 100 leaf pages. When the table grows beyond 10,000 rows, to a size between 10,001 and 1,000,000 rows, it requires a three-level index: the root page, a set of 100 branch pages, and a set of up to 10,000 leaf pages.

Index entries contained within leaf pages are sorted in key-value order. An index entry consists of a *key* and one or more *row pointers*. The key is a copy of the indexed columns from one row of data. A row pointer provides an address used to locate a row that contains the key. A unique index contains one index entry for every row in the table.

For information about special indexes for Dynamic Server, see "Indexes on User-Defined Data Types" on page 7-22.

To estimate the number of index pages

1. Add up the total widths of the indexed column or columns.

   This value is referred to as colsize. Add 4 to *colsize* to obtain *keysize*, the actual size of a key in the index.

   For example, if *colsize* is 6, the value of *keysize* is 10.

2. Calculate the expected proportion of unique entries to the total number of rows.

   The formulas in subsequent steps refer to this value as *propunique*.

   If the index is unique or has few duplicate values, use 1 for *propunique*.

   If a significant proportion of entries are duplicates, divide the number of unique index entries by the number of rows in the table to obtain a fractional value for *propunique*. For example, if the number of rows in the table is 4,000,000 and the number of unique index entries is 1,000,000, the value of *propunique* is .25.

   If the resulting value for *propunique* is less than .01, use .01 in the calculations that follow.

3. Estimate the size of a typical index entry with one of the following formulas, depending on whether the table is fragmented or not:

a. For nonfragmented tables, use the following formula:

```
entrysize = (keysize * propunique) + 5
```

The value 5 represents the number of bytes for the row pointer in a nonfragmented table.

For nonunique indexes, the database server stores the row pointer for each row in the index node but stores the key value only once. The entrysize value represents the average length of each index entry, even though some entries consist of only the row pointer. For example, if *propunique* is .25, the average number of rows for each unique key value is 4. If *keysize* is 10, the value of *entrysize* is 7.5. The following calculation shows the space required for all four rows:

```
space for four rows = 4 * 7.5 = 30
```

This space requirement is the same when you calculate it for the key value and add the four row pointers, as the following formula shows:

```
space for four rows = 10 + (4 * 5) = 30
```

b. For fragmented tables, use the following formula:

```
entrysize = (keysize * propunique) + 9
```

The value 9 represents the number of bytes for the row pointer in a fragmented table.

4. Estimate the number of entries per index page with the following formula:

```
pagents = trunc(pagefree/entrysize)
```

pagefree is the page size minus the page header (2020 for a 2-kilobyte page size).

entrysize is the value that you estimate in step 3.

The **trunc()** function notation indicates that you should round down to the nearest integer value.

5. Estimate the number of leaf pages with the following formula:

   ```
   leaves = ceiling(rows/pagents)
   ```

   rows is the number of rows that you expect to be in the table.

   pagents is the value that you estimate in step 4.

   The **ceiling()** function notation indicates that you should round up to the nearest integer value.

6. Estimate the number of branch pages at the second level of the index with the following formula:

   $branches_0$ = ceiling(*leaves/node_ents*)

   Calculate the value for *node_ents* with the following formula:

   ```
   node_ents = trunc( pagefree / ( keysize + 4) )
   ```

   pagefree is the same value as in step 4.

   keysize is the *colsize* plus 4 obtained in step 1.

   In the formula, 4 represents the number of bytes for the leaf node pointer.

7. If the value of $branches_0$ is greater than 1, more levels remain in the index.

   To calculate the number of pages contained in the next level of the index, use the following formula:

   $branches_{n+1}$ = ceiling($branches_n$/*node_ents*)

   $branches_n$ is the number of branches for the last index level that you calculated.

   $branches_{n+1}$ is the number of branches in the next level.

   node_ents is the value that you estimated in step 6.

8. Repeat the calculation in step 7 for each level of the index until the value of $branches_{n+1}$ equals 1.
9. Add up the total number of pages for all branch levels calculated in steps 6 through 8. This sum is called *branchtotal*.
10. Use the following formula to calculate the number of pages in the compact index:

    ```
    compactpages = (leaves + branchtotal)
    ```

11. If your database server instance uses a fill factor for indexes, the size of the index increases.

    The default fill factor value is 90 percent. You can change the fill factor value for all indexes with the FILLFACTOR configuration parameter. You can also change the fill factor for an individual index with the FILLFACTOR clause of the CREATE INDEX statement in SQL.

    To incorporate the fill factor into your estimate for index pages, use the following formula:

    ```
    indexpages = 100 * compactpages / FILLFACTOR
    ```

The preceding estimate is a guideline only. As rows are deleted and new ones are inserted, the number of index entries can vary within a page. This method for estimating index pages yields a conservative (high) estimate for most indexes. For a more precise value, build a large test index with real data and check its size with the **oncheck** utility.

Estimating Index Pages for Spatial and User-Defined Data

The database server uses the following types of indexes:

- B-tree
- R-tree
- Indexes that DataBlade modules provide

B-Tree Indexes

Dynamic Server uses a B-tree index for the following values:

- Columns that contain built-in data types (referred to as a *traditional B-tree index*)

  Built-in data types include CHARACTER, DATETIME, INTEGER, FLOAT, and so forth. For more information on built-in data types, refer to *Informix Guide to SQL: Reference*.

- Columns that contain one-dimensional user-defined data types (referred to as a *generic B-tree index*)

  User-defined data types include opaque and distinct data types. For more information on user-defined data types, refer to *Creating User-Defined Routines and User-Defined Data Types*.

- Values that a user-defined function returns (referred to as a *functional index*)

  The return value can be a built-in or user-defined data type but not a simple large object (TEXT or BYTE data type) or a smart large object (BLOB or CLOB data type). For more information on how to use functional indexes, refer to "Using a Functional Index" on page 7-28.

For information about how to estimate B-tree index size, refer to "Estimating Index Pages" on page 7-3.

R-Tree Indexes

Dynamic Server uses an R-tree index for spatial data (two-dimensional, three-dimensional, and so forth). For information about sizing an R-tree index, refer to the *Informix R-Tree Index User's Guide*.

Indexes That DataBlade Modules Provide

In Dynamic Server, users can access user-defined data types that DataBlade modules provide. A DataBlade module can also provide a user-defined index for the new data type. For example, the Excalibur Text Search Datablade provides an index to search text data. For more information, refer to the *Excalibur Text Search DataBlade Module User's Guide*.

For more information on the types of data and functions that each DataBlade module provides, refer to the user guide of each DataBlade module. For information on how to determine the types of indexes available in your database, see "Determining the Available Access Methods" on page 7-25.

Managing Indexes

An index is necessary on any column or combination of columns that must be unique. However, as discussed in Chapter 10, "Queries and the Query Optimizer," the presence of an index can also allow the query optimizer to speed up a query. The optimizer can use an index in the following ways:

- To replace repeated sequential scans of a table with nonsequential access
- To avoid reading row data when processing expressions that name only indexed columns
- To avoid a sort (including building a temporary table) when executing the GROUP BY and ORDER BY clauses

As a result, an index on the appropriate column can save thousands, tens of thousands, or in extreme cases, even millions of disk operations during a query. However, indexes entail costs.

Space Costs of Indexes

The first cost of an index is disk space. The presence of an index can add many pages to a dbspace; it is easy to have as many index pages as row pages in an indexed table. An estimating method appears in "Estimating lndex Pages" on page 7-3.

Time Costs of Indexes

The second cost of an index is time whenever the table is modified. The following descriptions assume that approximately two pages must be read to locate an index entry. That is the case when the index consists of a root page, one level of branch pages, and a set of leaf pages. The root page is assumed to be in a buffer already. The index for a very large table has at least two intermediate levels, so about three pages are read when the database server references such an index.

Presumably, one index is used to locate a row being altered. The pages for that index might be found in page buffers in shared memory for the database server. However, the pages for any other indexes that need altering must be read from disk.

Under these assumptions, index maintenance adds time to different kinds of modifications, as the following list shows:

- When you delete a row from a table, the database server must delete its entries from all indexes.

  The database server must look up the entry for the deleted row (two or three pages in) and rewrite the leaf page. The write operation to update the index is performed in memory, and the leaf page is flushed when the least recently used (LRU) buffer that contains the modified page is cleaned. This operation requires two or three page accesses to read the index pages if needed and one deferred page access to write the modified page.

- When you insert a row, the database server must insert its entries in all indexes.

  The database server must find a place in which to enter the inserted row within each index (two or three pages in) and rewrite (one deferred page out), for a total of three or four immediate page accesses per index.

- When you update a row, the database server must look up its entries in each index that applies to an altered column (two or three pages in).

  The database server must rewrite the leaf page to eliminate the old entry (one deferred page out) and then locate the new column value in the same index (two or three more pages in) and the row entered (one more deferred page out).

Insertions and deletions change the number of entries on a leaf page. Although virtually every *pagents* operation requires some additional work to deal with a leaf page that has either filled or been emptied, if *pagents* is greater than 100, this additional work occurs less than 1 percent of the time. You can often disregard it when you estimate the I/O impact.

In short, when a row is inserted or deleted at random, allow three to four added page I/O operations per index. When a row is updated, allow six to eight page I/O operations for each index that applies to an altered column. If a transaction is rolled back, all this work must be undone. For this reason, rolling back a transaction can take a long time.

Because the alteration of the row itself requires only two page I/O operations, index maintenance is clearly the most time-consuming part of data modification. For information about one way to reduce this cost, refer to "Clustering" on page 7-15.

Choosing Columns for Indexes

Indexes are required on columns that must be unique and are not specified as primary keys. In addition, add an index on columns that:

- Are used in joins that are not specified as foreign keys
- Are frequently used in filter expressions
- Are frequently used for ordering or grouping
- Do not involve duplicate keys
- Are amenable to clustered indexing

Filtered Columns in Large Tables

If a column is often used to filter the rows of a large table, consider placing an index on it. The optimizer can use the index to select the desired columns and avoid a sequential scan of the entire table. One example is a table that contains a large mailing list. If you find that a postal-code column is often used to filter a subset of rows, consider putting an index on that column.

This strategy yields a net savings of time only when the selectivity of the column is high; that is, when only a small fraction of rows holds any one indexed value. Nonsequential access through an index takes several more disk I/O operations than sequential access does, so if a filter expression on the column passes more than a fourth of the rows, the database server might as well read the table sequentially.

As a rule, indexing a filter column saves time in the following cases:

- The column is used in filter expressions in many queries or in slow queries.
- The column contains at least 100 unique values.
- Most column values appear in fewer than 10 percent of the rows.

Order-By and Group-By Columns

When a large quantity of rows must be ordered or grouped, the database server must put the rows in order. One way that the database server performs this task is to select all the rows into a temporary table and sort the table. But, as Chapter 10, "Queries and the Query Optimizer," discusses, if the ordering columns are indexed, the optimizer sometimes reads the rows in sorted order through the index, thus avoiding a final sort.

Because the keys in an index are in sorted sequence, the index really represents the result of sorting the table. By placing an index on the ordering column or columns, you can replace many sorts during queries with a single sort when the index is created.

Avoiding Columns with Duplicate Keys

When duplicate keys are permitted in an index, entries that match a given key value are grouped in lists. The database server uses these lists to locate rows that match a requested key value. When the selectivity of the index column is high, these lists are generally short. But when only a few unique values occur, the lists become long and can cross multiple leaf pages.

Placing an index on a column that has low selectivity (that is, a small number of distinct values relative to the number of rows) can reduce performance. In such cases, the database server must not only search the entire set of rows that match the key value, but it must also lock all the affected data and index pages. This process can impede the performance of other update requests as well.

To correct this problem, replace the index on the low-selectivity column with a composite index that has a higher selectivity. Use the low-selectivity column as the leading column and a high-selectivity column as your second column in the index. The composite index limits the number of rows that the database server must search to locate and apply an update.

You can use any second column to disperse the key values as long as its value does not change, or changes at the same time as the real key. The shorter the second column the better, because its values are copied into the index and expand its size.

Clustering

Clustering is a method for arranging the rows of a table so that their physical order on disk closely corresponds to the sequence of entries in the index. (Do not confuse the clustered index with an *optical cluster*, which is a method for storing logically related TEXT or BYTE data together on an optical volume.)

When you know that a table is ordered by a certain index, you can avoid sorting. You can also be sure that when the table is searched on that column, it is read effectively in sequential order, instead of nonsequentially. These points are covered in Chapter 10, "Queries and the Query Optimizer."

Tip: *For information about eliminating interleaved extents by altering an index to cluster, see "Creating or Altering an Index to Cluster" on page 6-43.*

In the **stores_demo** database, the **orders** table has an index, **zip_ix,** on the postal-code column. The following statement causes the database server to put the rows of the **customer** table in descending order by postal code:

```
ALTER INDEX zip_ix TO CLUSTER
```

To cluster a table on a nonindexed column, you must create an index. The following statement reorders the **orders** table by order date:

```
CREATE CLUSTERED INDEX o_date_ix ON orders (order_date ASC)
```

To reorder a table, the database server must copy the table. In the preceding example, the database server reads all the rows in the table and constructs an index. Then it reads the index entries in sequence. For each entry, it reads the matching row of the table and copies it to a new table. The rows of the new table are in the desired sequence. This new table replaces the old table.

Clustering is not preserved when you alter a table. When you insert new rows, they are stored physically at the end of the table, regardless of their contents. When you update rows and change the value of the clustering column, the rows are written back into their original location in the table.

Clustering can be restored after the order of rows is disturbed by ongoing updates. The following statement reorders the table to restore data rows to the index sequence:

```
ALTER INDEX o_date_ix TO CLUSTER
```

Reclustering is usually quicker than the original clustering because reading out the rows of a nearly clustered table is similar in I/O impact to a sequential scan.

Clustering and reclustering take a lot of space and time. To avoid some clustering, build the table in the desired order initially.

Dropping Indexes

In some applications, most table updates can be confined to a single time period. You might be able to set up your system so that all updates are applied overnight or on specified dates.

When updates are performed as a batch, you can drop all nonunique indexes while you make updates and then create new indexes afterward. This strategy can have the following positive effects:

- The updating program can run faster with fewer indexes to update. Often, the total time to drop the indexes, update without them, and re-create them is less than the time to update with the indexes in place. (For a discussion of the time cost of updating indexes, refer to "Time Costs of Indexes" on page 7-12.)
- Newly made indexes are more efficient. Frequent updates tend to dilute the index structure so that it contains many partly full leaf pages. This dilution reduces the effectiveness of an index and wastes disk space.

As a time-saving measure, make sure that a batch-updating program calls for rows in the sequence that the primary-key index defines. That sequence causes the pages of the primary-key index to be read in order and only one time each.

The presence of indexes also slows down the population of tables when you use the LOAD statement or the **dbload** utility. Loading a table that has no indexes is a quick process (little more than a disk-to-disk sequential copy), but updating indexes adds a great deal of overhead.

To load a table that has no indexes

1. Drop the table (if it exists).
2. Create the table without specifying any unique constraints.
3. Load all rows into the table.
4. Alter the table to apply the unique constraints.
5. Create the nonunique indexes.

If you cannot guarantee that the loaded data satisfies all unique constraints, you must create unique indexes before you load the rows. It saves time if the rows are presented in the correct sequence for at least one of the indexes. If you have a choice, make it the row with the largest key. This strategy minimizes the number of leaf pages that must be read and written.

Improving Performance for Index Builds

Whenever possible, the database server uses parallel processing to improve the response time of index builds. The number of parallel processes is based on the number of fragments in the index and the value of the **PSORT_NPROCS** environment variable. The database server builds the index with parallel processing even when the value of PDQ priority is `0`.

You can often improve the performance of an index build by taking the following steps:

1. Set PDQ priority to a value greater than `0` to obtain more memory than the default 128 kilobytes.

   When you set PDQ priority to greater than `0`, the index build can take advantage of the additional memory for parallel processing.

   To set PDQ priority, use either the **PDQPRIORITY** environment variable or the SET PDQPRIORITY statement in SQL.

2. Do not set the **PSORT_NPROCS** environment variable.

   Informix recommends that you not set the **PSORT_NPROCS** environment variable. If you have a computer with multiple CPUs, the database server uses two threads per sort when it sorts index keys and **PSORT_NPROCS** is not set. The number of sorts depends on the number of fragments in the index, the number of keys, the key size, and the values of the PDQ memory configuration parameters.

3. Allocate enough memory and temporary space to build the entire index.

   a. Estimate the amount of virtual shared memory that the database server might need for sorting.

      For more information, refer to "Estimating Sort Memory" on page 7-19.

   b. Specify more memory with the DS_TOTAL_MEMORY and DS_MAX_QUERIES configuration parameters.

   c. If not enough memory is available, estimate the amount of temporary space needed for an entire index build.

      For more information, refer to "Estimating Temporary Space for Index Builds" on page 7-20.

   d. Use the **onspaces -t** utility to create large temporary dbspaces and specify them in the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable.

      For information on how to optimize temporary dbspaces, refer to "Configuring Dbspaces for Temporary Tables and Sort Files" on page 5-12.

Estimating Sort Memory

To calculate the amount of virtual shared memory that the database server might need for sorting, estimate the maximum number of sorts that might occur concurrently and multiply that number by the average number of rows and the average row size.

For example, if you estimate that 30 sorts could occur concurrently, the average row size is 200 bytes, and the average number of rows in a table is 400, you can estimate the amount of shared memory that the database server needs for sorting as follows:

```
30 sorts * 200 bytes * 400 rows = 2,400,000 bytes
```

If PDQ priority is `0`, the maximum amount of shared memory that the database server allocates for a sort is about 128 kilobytes.

If PDQ priority is greater than 0, the maximum amount of shared memory that the database server allocates for a sort is controlled by the memory grant manager (MGM). The MGM uses the settings of PDQ priority and the following configuration parameters to determine how much memory to grant for the sort:

- DS_TOTAL_MEMORY
- DS_MAX_QUERIES
- MAX_PDQPRIORITY

For more information about allocating memory for parallel processing, refer to "Allocating Resources for Parallel Database Queries" on page 12-12.

Estimating Temporary Space for Index Builds

To estimate the amount of temporary space needed for an entire index build, perform the following steps:

1. Add up the total widths of the indexed columns or returned values from user-defined functions. This value is referred to as *keysize*.
2. Estimate the size of a typical item to sort with one of the following formulas, depending on whether the index is attached or not:
   a. For a nonfragmented table and a fragmented table with an index created without an explicit fragmentation strategy, use the following formula:
      ```
      sizeof_sort_item = keysize + 4
      ```
   b. For fragmented tables with the index explicitly fragmented, use the following formula:
      ```
      sizeof_sort_item = keysize + 8
      ```
3. Estimate the number of bytes needed to sort with the following formula:
   ```
   temp_bytes = 2 * (rows * sizeof_sort_item)
   ```
   This formula uses the factor 2 because everything is stored twice when intermediate sort runs use temporary space. Intermediate sort runs occur when not enough memory exists to perform the entire sort in memory.

   The value for *rows* is the total number of rows that you expect to be in the table.

Improving Performance for Index Checks

The **oncheck** utility provides better concurrency for tables that use row locking. When a table uses page locking, **oncheck** places a shared lock on the table when it performs index checks. Shared locks do not allow other users to perform updates, inserts, or deletes on the table while **oncheck** checks or prints the index information.

If the table uses page locking, the database server returns the following message when you run **oncheck** with the **-x** option:

```
WARNING: index check requires an s-lock on tables whose lock level
is page.
```

For a detailed description of **oncheck** locking and the **-x** option, refer to the utilities chapter in the *Informix Administrator's Reference.*

The following summary describes locking performed during index checks:

- By default, the database server does not place a shared lock on the table when you check an index with the **oncheck -ci, -cI, -pk, -pK, -pl,** or **-pL** options unless the table uses page locking. When **oncheck** checks indexes for a table with page locking, it places a shared lock on the table, so no other users can perform updates, inserts, or deletes until the check has completed.
- By not placing a shared lock on tables using row locks during index checks, the **oncheck** utility cannot be as accurate in the index check. For absolute assurance of a complete index check, execute **oncheck** with the **-x** option. With the **-x** option, **oncheck** places a shared lock on the table, and no other users can perform updates, inserts, or deletes until the check completes.

You can query the **systables** system catalog table to see the current lock level of the table, as the following sample SQL statement shows:

```
SELECT locklevel FROM systables
   WHERE tabname = "customer"
```

If you do not see a value of R (for row) in the **locklevel** column, you can modify the lock level, as the following sample SQL statement shows:

```
ALTER TABLE tab1 LOCK MODE (ROW);
```

Row locking might add other side effects, such as an overall increase in lock usage. For more information about locking levels, refer to Chapter 8, "Locking."

Indexes on User-Defined Data Types

Users can define their own data types and the functions that operate on these data types. DataBlade modules also provide extended data types and functions to the database server. You can define indexes on the following kinds of user-defined data types:

- Opaque data types

  An *opaque data type* is a fundamental data type that you can use to define columns in the same way you use built-in types. An opaque data type stores a single value and cannot be divided into components by the database server. For information about creating opaque data types, see the CREATE OPAQUE TYPE statement in the *Informix Guide to SQL: Syntax* and *Creating User-Defined Routines and User-Defined Data Types*. For more information on the data types and functions that each DataBlade module provides, refer to the user guide of each DataBlade module.

- Distinct data types

  A *distinct data type* has the same representation as an existing opaque or built-in data type but is different from these types. For information about distinct data types, see the *Informix Guide to SQL: Reference* and the CREATE DISTINCT TYPE statement in the *Informix Guide to SQL: Syntax*.

For more information on data types, refer to the *Informix Guide to SQL: Reference*.

Defining Indexes for User-Defined Data Types

As with built-in data types, you might improve the response time for a query when you define indexes for new data types. The response time for a query might improve when Dynamic Server uses an index for:

- Columns used to join two tables
- Columns that are filters for a query
- Columns in an ORDER BY or GROUP BY clause
- Results of functions that are filters for a query

For more information on when the query performance can improve with an index on a built-in data type, refer to "Improving Performance with Indexes" on page 13-23.

Dynamic Server and DataBlade modules provide a variety of different types of indexes (also referred to as *secondary-access methods*). A secondary-access method is a set of database server functions that build, access, and manipulate an index structure. These functions encapsulate index operations, such as how to scan, insert, delete, or update nodes in an index.

To create an index on a user-defined data type, you can use any of the following secondary-access methods:

- Generic B-tree index

  A B-tree index is good for a query that retrieves a range of data values. For more information, refer to "B-Tree Secondary-Access Method" on page 7-24.

- R-tree index

  An R-tree index is good for searches on multidimensional data. For more information, refer to the *Informix R-Tree Index User's Guide*.

- Secondary-access methods that a DataBlade module provides for a new data type

  A DataBlade module that supports a certain type of data can also provide a new index for that new data type. For more information, refer to "Using an Index That a DataBlade Module Provides" on page 7-31.

You can create a functional index on the resulting values of a user-defined function on one or more columns. For more information, see "Using a Functional Index" on page 7-28.

Once you choose the desired index type, you might also need to extend an operator class for the secondary-access method. For more information on how to extend operator classes, refer to the *Creating User-Defined Routines and User-Defined Data Types* manual.

B-Tree Secondary-Access Method

Dynamic Server provides the *generic B-tree index* for columns in database tables. In traditional relational database systems, the B-tree access method handles only built-in data types and therefore it can only compare two keys of built-in data types. The generic B-tree index is an extended version of a B-tree that Dynamic Server provides to support user-defined data types.

Tip: *For more information on the structure of a B-tree index and how to estimate the size of a B-tree index, refer to* *"Estimating Index Pages" on page 7-3***.**

Dynamic Server uses the generic B-tree as the built-in secondary-access method. This built-in secondary-access method is registered in the **sysams** system catalog table with the name **btree**. When you use the CREATE INDEX statement (without the USING clause) to create an index, the database server creates a generic B-tree index. For more information, see the CREATE INDEX statement in the *Informix Guide to SQL: Syntax*.

Tip: *Dynamic Server also defines another secondary-access method, the R-tree index. For more information on how to use an R-tree index, see the "Informix R-Tree Index User's Guide."*

Uses for a B-Tree Index

A B-tree index is good for a query that retrieves a range of data values. If the data to be indexed has a logical sequence to which the concepts of *less than*, *greater than*, and *equal* apply, the generic B-tree index is a useful way to index your data. Initially, the generic B-tree index supports the relational operators (<,<=,=,>=,>) on all built-in data types and orders the data in lexicographical sequence.

The optimizer considers whether to use the B-tree index to execute a query if you define a generic B-tree index on:

- Columns used to join two tables
- Columns that are filters for a query
- Columns in an ORDER BY or GROUP BY clause
- Results of functions that are filters for a query

Extending a Generic B-Tree Index

Initially, the generic B-tree can index data that is one of the built-in data types, and it orders the data in lexicographical sequence. However, you can extend a generic B-tree to support columns and functions on the following data types:

- *User-defined data types* (opaque and distinct data types) that you want the B-tree index to support

  In this case, you need to extend the default operator class of the generic B-tree index.

- *Built-in data types* that you want to order in a different sequence from the lexicographical sequence that the generic B-tree index uses

  In this case, you need to define a different operator class from the default generic B-tree index.

An *operator class* is the set of functions (operators) that are associated with a nontraditional B-tree index. For more details on operator classes, refer to "Choosing Operator Classes for Indexes" on page 7-31.

Determining the Available Access Methods

Dynamic Server provides a built-in B-tree secondary-access method. Your environment might have installed DataBlade modules that implement additional secondary-access methods. If additional access methods exist, they are defined in the **sysams** system catalog table.

To determine the secondary-access methods that are available for your database, query the **sysams** system catalog table with the following SELECT statement:

```
SELECT am_id, am_owner, am_name, am_type FROM sysams
   WHERE am_type = 'S';
```

An 'S' value in the **am_type** column identifies the access method as a secondary-access method. This query returns the following information:

- The **am_id** and **am_name** columns identify the secondary-access method.
- The **am_owner** column identifies the owner of the access method.

In an ANSI-compliant database, the access-method name must be unique within the name space of the user. The access-method name always begins with the owner in the format **am_owner.am_name**.

By default, Dynamic Server provides the following definitions in the **sysams** system catalog table for two secondary-access methods, **btree** and **rtree**.

Access Method	am_id Column	am_name Column	am_owner Column
Generic B-tree	1	btree	'informix'
R-tree	2	rtree	'informix'

Important: *The **sysams** system catalog table does not contain a row for the built-in primary access method. This primary access method is internal to Dynamic Server and does not require a definition in **sysams**. However, the built-in primary access method is always available for use.*

If you find additional rows in the **sysams** system catalog table (rows with **am_id** values greater than 2), the database supports additional user-defined access methods. Check the value in the **am_type** column to determine whether a user-defined access method is a primary- or secondary-access method.

For more information on the columns of the **sysams** system catalog table, see the *Informix Guide to SQL: Reference*. For information on how to determine the operator classes that are available in your database, see "Determining the Available Operator Classes" on page 7-35.

Using a User-Defined Secondary-Access Method

The built-in secondary-access method is a B-tree index. If the concepts of *less than*, *greater than*, and *equal* do not apply to the data to be indexed, you probably want to consider a *user-defined secondary-access method* that works with Dynamic Server. You can use a user-defined secondary-access method to access other indexing structures, such as an R-tree index.

If your database supports a user-defined secondary-access method, you can specify that the database server uses this access method when it accesses a particular index. For information on how to determine the secondary-access methods that your database defines, see "Determining the Available Access Methods" on page 7-25.

To choose a user-defined secondary-access method, use the USING clause of the CREATE INDEX statement. The USING clause specifies the name of the secondary-access method to use for the index you create. This name must be listed in the **am_name** column of the **sysams** system catalog table and must be a secondary-access method (the **am_type** column of **sysams** is `'S'`).

The secondary-access method that you specify in the USING clause of CREATE INDEX must already be defined in the **sysams** system catalog. If the secondary-access method has not yet been defined, the CREATE INDEX statement fails.

When you omit the USING clause from the CREATE INDEX statement, the database server uses B-tree indexes as the secondary-access method. For more information, see the CREATE INDEX statement in the *Informix Guide to SQL: Syntax*.

R-Tree Index

Dynamic Server supports the *R-tree index* for columns that contain spatial data such as maps and diagrams. An R-tree index uses a tree structure whose nodes store pointers to lower-level nodes. At the leaves of the R-tree are a collection of data pages that store *n*-dimensional shapes. For more information on the structure of an R-tree index and how to estimate the size of an R-tree index, refer to the *Informix R-Tree Index User's Guide*.

Using a Functional Index

Dynamic Server provides support for indexes on the following database objects:

- A column index

  You can create a column index on the actual values in one or more columns.

- A functional index

  You can create a functional index on the functional value of one or more columns.

Important: *You cannot create a functional index on the functional value of a column that contains a collection data type.*

To decide whether to use a column index or functional index, determine whether a column index is the right choice for the data that you want to index. An index on a column of some data types might not be useful for typical queries. For example, the following query asks how many images are dark:

```
SELECT COUNT(*) FROM photos WHERE darkness(picture) > 0.5
```

An index on the **picture** data itself does not improve the query performance. The concepts of *less than*, *greater than*, and *equal* are not particularly meaningful when applied to an image data type. Instead, a functional index that uses the **darkness()** function can improve performance. You might also have a user-defined function that executes frequently enough that performance improves when you create an index on its values.

What Is a Functional Index?

When you create a functional index, the database server computes the values of the user-defined function and stores them as key values in the index. When a change in the table data causes a change in one of the values of an index key, the database server automatically updates the functional index.

You can use a functional index for functions that return values of both user-defined data types (opaque and distinct) and built-in data types. However, you cannot define a functional index if the function returns a simple-large-object data type (TEXT or BYTE).

When Is a Functional Index Used?

The optimizer considers whether to use a functional index to access the results of functions that are in one of the following query clauses:

- SELECT clause
- Filters in the WHERE clause

A functional index can be a B-tree index, an R-tree index, or a user-defined index type that a DataBlade module provides. For more information on the types of indexes, see "Defining Indexes for User-Defined Data Types" on page 7-23. For information on space requirements for functional indexes, refer to "Estimating Index Pages" on page 3-13.

How Do You Create a Functional Index?

The function can be built in or user defined. A user-defined function can be either an external function or an SPL function.

To build a functional index on a user-defined function

1. Write the code for the user-defined function if it is an external function.
2. Register the user-defined function in the database with the CREATE FUNCTION statement.
3. Build the functional index with the CREATE INDEX statement.

To create a functional index on the darkness() function

1. Write the code for the user-defined **darkness()** function that operates on the data type and returns a decimal value.
2. Register the user-defined function in the database with the CREATE FUNCTION statement:

```
CREATE FUNCTION darkness(im image)
RETURNS decimal
EXTERNAL NAME '/lib/image.so'
LANGUAGE C NOT VARIANT
```

   In this example, you can use the default operator class for the functional index because the return value of the **darkness()** function is a built-in data type, DECIMAL.
3. Build the functional index with the CREATE INDEX statement.

```
CREATE TABLE photos
(
   name char(20),
   picture image
...
);
CREATE INDEX dark_ix ON photos (darkness(picture));
```

   In this example, assume that the user-defined data type of **image** has already been defined in the database.

The optimizer can now consider the functional index when you specify the **darkness()** function as a filter in the query:

```
SELECT count(*) FROM photos WHERE darkness(picture) > 0.5
```

You can also create a composite index with user-defined functions. For more information, see "Using Composite Indexes" on page 13-23.

Using an Index That a DataBlade Module Provides

In Dynamic Server, users can access new data types that DataBlade modules provide. A DataBlade module can also provide a new index for the new data type. For example, the Excalibur Text Search Datablade module provides an index to search text data. For more information, refer to the *Excalibur Text Search DataBlade Module User's Guide*.

For more information on the types of data and functions that each DataBlade module provides, refer to the user guide for the DataBlade module. For information on how to determine the types of indexes available in your database, see "Determining the Available Access Methods" on page 7-25.

Choosing Operator Classes for Indexes

For most situations, use the default operators that are defined for a secondary-access method. However, when you want to order the data in a different sequence or provide index support for a user-defined data type, you must extend an operator class. For more information on how to extend an operator class, refer to *Creating User-Defined Routines and User-Defined Data Types*.

Operator Classes

An *operator class* is a set of function names that is associated with a secondary-access method. These functions allow the secondary-access method to store and search for values of a particular data type. The query optimizer for the database server uses an operator class to determine if an index can process the query with the least cost. An operator class indicates two things to the query optimizer:

- Which functions that appear in an SQL statement can be evaluated with a given index

  These functions are called the *strategy functions* for the operator class.
- Which functions the index uses to evaluate the strategy functions

  These functions are called the *support functions* for the operator class.

With the information that the operator class provides, the query optimizer can determine whether a given index is applicable to the query. The query optimizer can consider whether to use the index for the given query when the following conditions are true:

- An index exists on the particular column or columns in the query.
- For the index that exists, the operation on the column or columns in the query matches one of the strategy functions in the operator class associated with the index.

The query optimizer reviews the available indexes for the table or tables and matches the index keys with the column specified in the query filter. If the column in the filter matches an index key, and the function in the filter is one of the strategy functions of the operator class, the optimizer includes the index when it determines which query plan has the lowest execution cost. In this manner, the optimizer can determine which index can process the query with the least cost.

Dynamic Server stores information about operator classes in the **sysopclasses** system catalog table.

Strategy and Support Functions

Dynamic Server uses the *strategy functions* of a secondary-access method to help the query optimizer determine whether a specific index is applicable to a specific operation on a data type. If an index exists and the operator in the filter matches one of the strategy functions in the operator class, the optimizer considers whether to use the index for the query.

Dynamic Server uses the *support functions* of a secondary-access method to build and access the index. These functions are not called directly by end users. When an operator in the query filter matches one of the strategy functions, the secondary-access method uses the support functions to traverse the index and obtain the results. Identification of the actual support functions is left to the secondary-access method.

Default Operator Classes

Each secondary-access method has a *default operator class* associated with it. By default, the CREATE INDEX statement associates the default operator class with an index. For example, the following CREATE INDEX statement creates a B-tree index on the **postalcode** column and automatically associates the default B-tree operator class with this column:

```
CREATE INDEX postal_ix ON customer(postalcode)
```

For more information on how to specify a new default operator class for an index, see "Using an Operator Class" on page 7-36.

Built-In B-Tree Operator Class

The built-in secondary-access method, the generic B-tree, has a default operator class called **btree_ops** defined in the **sysopclasses** system catalog table. By default, the CREATE INDEX statement associates the **btree_ops** operator class with it when you create a B-tree index. For example, the following CREATE INDEX statement creates a generic B-tree index on the **order_date** column of the **orders** table and associates with this index the default operator class for the B-tree secondary-access method:

```
CREATE INDEX orddate_ix ON orders (order_date)
```

Dynamic Server uses the **btree_ops** operator class to specify:

- The strategy functions to tell the optimizer which filters in a query can use a B-tree index
- The support function to build and search the B-tree index

B-Tree Strategy Functions

The **btree_ops** operator class defines the following names of strategy functions for the **btree** access method:

- **lessthan** (<)
- **lessthanorequal** (<=)
- **equal** (=)
- **greaterthanorequal** (>=)
- **greaterthan** (>)

These strategy functions are all *operator functions*. That is, each function is associated with an operator symbol; in this case, with a relational-operator symbol. For more information on relational-operator functions, see *Creating User-Defined Routines and User-Defined Data Types*.

When the query optimizer examines a query that contains a column, it checks to see if this column has a B-tree index defined on it. If such an index exists *and* if the query contains one of the relational operators that the **btree_ops** operator class supports, the optimizer can choose a B-tree index to execute the query.

B-Tree Support Function

The **btree_ops** operator class has one support function, a comparison function called **compare()**. The **compare()** function is a user-defined function that returns an integer value to indicate whether its first argument is equal to, less than, or greater than its second argument, as follows:

- A value of `0` when the first argument is *equal to* the second argument
- A value less than `0` when the first argument is *less than* the second argument
- A value greater than `0` when the first argument is *greater than* the second argument

The B-tree secondary-access method uses the **compare()** function to traverse the nodes of the generic B-tree index. To search for data values in a generic B-tree index, the secondary-access method uses the **compare()** function to compare the key value in the query to the key value in an index node. The result of the comparison determines if the secondary-access method needs to search the next-lower level of the index or if the key resides in the current node.

The generic B-tree access method also uses the **compare()** function to perform the following tasks for generic B-tree indexes:

- Sort the keys before the index is built
- Determine the linear order of keys in a generic B-tree index
- Evaluate the relational operators
- Search for data values in an index

The database server uses the **compare()** function to evaluate comparisons in the SELECT statement. To provide support for these comparisons for opaque data types, you must write the **compare()** function. For more information, see *Creating User-Defined Routines and User-Defined Data Types*.

The database server also uses the **compare()** function when it uses a B-tree index to process an ORDER BY clause in a SELECT statement. However, the optimizer does not use the index to perform an ORDER BY operation if the index does not use the btree-ops operator class.

Determining the Available Operator Classes

The database server provides the default operator class for the built-in secondary-access method, the generic B-tree index. In addition, your environment might have installed DataBlade modules that implement other operator classes. All operator classes are defined in the **sysopclasses** system catalog table.

To determine the operator classes that are available for your database, query the **sysopclasses** system catalog table with the following SELECT statement:

```
SELECT opclassid, opclassname, amid, am_name
FROM sysopclasses, sysams
WHERE sysopclasses.amid = sysams.am_id
```

This query returns the following information:

- The **opclassid** and **opclassname** columns identify the operator class.
- The **am_id** and **am_name** columns identify the associated secondary-access methods.

By default, the database server provides the following definitions in the **sysopclasses** system catalog table for two operator classes, **btree_ops** and **rtree_ops**.

Access Method	opclassid Column	opclassname Column	amid Column	am_name Column
Generic B-tree	1	btree_ops	1	btree
R-tree	2	rtree_ops	2	rtree

If you find additional rows in the **sysopclasses** system catalog table (rows with **opclassid** values greater than 2), your database supports user-defined operator classes. Check the value in the **amid** column to determine the secondary-access methods to which the operator class belongs.

The **am_defopclass** column in the **sysams** system catalog table stores the operator-class identifier for the default operator class of a secondary-access method. To determine the default operator class for a given secondary-access method, you can run the following query:

```
SELECT am_id, am_name, am_defopclass, opclass_name
FROM sysams, sysopclasses
WHERE sysams.am_defopclass = sysopclasses.opclassid
```

By default, the database server provides the following default operator classes.

Access Method	am_id Column	am_name Column	am_defopclass Column	opclass_name Column
Generic B-tree	1	btree	1	btree_ops
R-tree	2	rtree	2	rtree_ops

For more information on the columns of the **sysopclasses** and **sysams** system catalog tables, see the *Informix Guide to SQL: Reference*. For information on how to determine the access methods that are available in your database, see "Determining the Available Access Methods" on page 7-25.

Using an Operator Class

The CREATE INDEX statement specifies the operator class to use for each component of an index. If you do not specify an operator class, CREATE INDEX uses the default operator class for the secondary-access method that you create. You can use a user-defined operator class for components of an index. To specify a user-defined operator class for a particular component of an index, you can:

- Use a user-defined operator class that your database already defines.
- Use an R-tree operator class, if your database defined the R-tree secondary-access method. For more information about R-trees, refer to the *Informix R-Tree Index User's Guide*.

If your database supports multiple-operator classes for the secondary-access method that you want to use, you can specify which operator classes the database server is to use for a particular index. For information on how to determine the operator classes that your database defines, see "Determining the Available Operator Classes" on page 7-35.

Each part of a composite index can specify a different operator class. You choose the operator classes when you create the index. In the CREATE INDEX statement, you specify the name of the operator class to use after each column or function name in the index-key specification. Each name must be listed in the **opclassname** column of the **sysopclasses** system catalog table and must be associated with the secondary-access method that the index uses.

For example, if your database defines the **abs_btree_ops** secondary-access method to define a new sort order, the following CREATE INDEX statement specifies that the **table1** table associates the **abs_btree_ops** operator class with the **col1_ix** B-tree index:

```
CREATE INDEX col1_ix ON table1(col1 abs_btree_ops)
```

The operator class that you specify in the CREATE INDEX statement must already be defined in the **sysopclasses** system catalog with the CREATE OPCLASS statement. If the operator class has not yet been defined, the CREATE INDEX statement fails. For information on how to create an operator class, refer to *Creating User-Defined Routines and User-Defined Data Types.*

Chapter

8

Locking

In This Chapter 8-3

Lock Granularity 8-3
Row and Key Locks 8-4
Advantages and Disadvantages of Row and Key Locks . . . 8-4
Key-Value Locks 8-4
Page Locks 8-5
Table Locks 8-6
Database Locks 8-7

Configuring Lock Mode 8-7

Lock Waits 8-9

Locks with the SELECT Statement 8-9
Isolation Level 8-9
Dirty Read Isolation 8-10
Committed Read Isolation 8-10
Cursor Stability Isolation 8-11
Repeatable Read Isolation 8-11
Locking Nonlogging Tables 8-12
Update Cursors. 8-13

Locks Placed with INSERT, UPDATE, and DELETE 8-15

Monitoring and Administering Locks 8-15
Monitoring Locks 8-16
Configuring and Monitoring the Number of Locks 8-18
Monitoring Lock Waits and Lock Errors 8-19
Monitoring Deadlocks 8-21
Monitoring Isolation That Sessions Use 8-22

Locks for Smart Large Objects 8-22
Types of Locks on Smart Large Objects 8-23
Byte-Range Locking 8-23
How the Database Server Manages Byte-Range Locks 8-24
Using Byte-Range Locks 8-24
Monitoring Byte-Range Locks 8-25
Setting Number of Locks for Byte-Range Locking 8-26
Lock Promotion 8-27
Dirty Read and Smart Large Objects 8-28

In This Chapter

This chapter describes how the database server uses locks, how locks can affect concurrency and performance, and how to monitor and administer locks.

Lock Granularity

A *lock* is a software mechanism that prevents others from using a resource. This chapter discusses the locking mechanism placed on data. You can place a lock on the following items:

- An individual row or key
- A page of data or index keys
- A table
- A database

Additional types of locks are available for smart large objects. For more information, refer to "Locks for Smart Large Objects" on page 8-22.

The level and type of information that the lock protects is called *locking granularity*. Locking granularity affects performance. When a user cannot access a row or key, the user can wait for another user to unlock the row or key. If a user locks an entire page, a higher probability exists that more users will wait for a row in the page. The ability of more than one user to access a set of rows is called *concurrency*. The goal of the database administrator is to increase concurrency to increase total performance without sacrificing performance for an individual user.

Row and Key Locks

Row and key locking are not the default behaviors. The default locking mode is page-locking, as "Page Locks" on page 8-5 explains. You must create the table with row-level locking on, as in the following example:

```
CREATE TABLE customer(customer_num serial, lname char(20)...)
   LOCK MODE ROW;
```

The ALTER TABLE statement can also change the lock mode.

When the lock mode is ROW and you insert or update a row, the database server creates a row lock. In some cases, you place a row lock by simply reading the row with a SELECT statement.

When the lock mode is ROW and you insert, update, or delete a key (performed automatically when you insert, update, or delete a row), the database server creates a lock on the key in the index.

Advantages and Disadvantages of Row and Key Locks

Row and key locks generally provide the best overall performance when you are updating a relatively small number of rows because they increase concurrency. However, the database server incurs some overhead in obtaining a lock. For an operation that changes a large number of rows, obtaining one lock per row might not be cost effective. In that case, consider page locking.

Key-Value Locks

When a user deletes a row within a transaction, the row cannot be locked because it does not exist. However, the database server must somehow record that a row existed until the end of the transaction.

The database server uses a concept called *key-value locking* to lock the deleted row. When the database server deletes a row, key values in the indexes for the table are not removed immediately. Instead, each key value is marked as deleted, and a lock is placed on the key value.

Other users might encounter key values that are marked as deleted. The database server must determine whether a lock exists. If a lock exists, the delete has not been committed, and the database server sends a lock error back to the application (or it waits for the lock to be released if the user executed SET LOCK MODE TO WAIT).

One of the most important uses for key-value locking is to assure that a unique key remains unique through the end of the transaction that deleted it. Without this protection mechanism, user A might delete a unique key within a transaction, and user B might insert a row with the same key before the transaction commits. This scenario makes rollback by user A impossible. Key-value locking prevents user B from inserting the row until the end of user A's transaction.

Page Locks

Page locking is the default behavior when you create a table without the LOCK MODE clause. With page locking, instead of locking only the row, the database server locks the entire page that contains the row. If you update several rows on the same page, the database server uses only one lock for the page.

When you insert or update a row, the database server creates a page lock on the data page. In some cases, the database server creates a page lock when you simply read the row with a SELECT statement.

When you insert, update, or delete a key (performed automatically when you insert, update, or delete a row), the database server creates a lock on the page that contains the key in the index.

Important: *A page lock on an index page can decrease concurrency more substantially than a page lock on a data page. Index pages are dense and hold a large number of keys. By locking an index page, you make a potentially large number of keys unavailable to other users until you release the lock.*

Page locks are useful for tables in which the normal user changes a large number of rows at one time. For example, an orders table that holds orders that are commonly inserted and queried individually is not a good candidate for page locking. But a table that holds old orders and is updated nightly with all of the orders placed during the day might be a good candidate. In this case, the type of isolation level that you use to access the table is important. For more information, refer to "Isolation Level" on page 8-9.

Table Locks

In a data warehouse environment, it might be more appropriate for queries to acquire locks of larger granularity. For example, if a query accesses most of the rows in a table, its efficiency increases if it acquires a smaller number of table locks instead of many page or row locks.

The database server can place two types of table locks:

- Shared lock

  No other users can write to the table.

- Exclusive lock

  No other users can read from or write to the table.

Another important distinction between these two types of table locks is the actual number of locks placed:

- In shared mode, the database server places one shared lock on the table, which informs other users that no updates can be performed. In addition, the database server adds locks for every row updated, deleted, or inserted.
- In exclusive mode, the database server places only one exclusive lock on the table, no matter how many rows it updates. If you update most of the rows in the table, place an exclusive lock on the table.

Important: *A table lock on a table can decrease update concurrency radically. Only one update transaction can access that table at any given time, and that update transaction locks out all other transactions. However, multiple read-only transactions can simultaneously access the table. This behavior is useful in a data warehouse environment where the data is loaded and then queried by multiple users.*

You can switch a table back and forth between table-level locking and the other levels of locking. This ability to switch locking levels is useful when you use a table in a data warehouse mode during certain time periods but not in others.

A transaction tells the database server to use table-level locking for a table with the LOCK TABLE statement. The following example places an exclusive lock on the table:

```
LOCK TABLE tab1 IN EXCLUSIVE MODE;
```

The following example places a shared lock on the table:

```
LOCK TABLE tab1 IN SHARE MODE:
```

In some cases, the database server places its own table locks. For example, if the isolation level is Repeatable Read, and the database server has to read a large portion of the table, it places a table lock automatically instead of setting row or page locks. The database server places a table lock on a table when it creates or drops an index.

Database Locks

You can place a lock on the entire database when you open the database with the DATABASE statement. A database lock prevents read or update access by anyone but the current user.

The following statement opens and locks the sales database:

```
DATABASE sales EXCLUSIVE
```

Configuring Lock Mode

When you create a table, the default lock mode is page. If you have many users accessing the same tables, you can increase concurrency by using a smaller locking granularity. If you know that most of your applications might benefit from a lock mode of row, you can take one of the following actions:

- Use the LOCK MODE ROW clause in each CREATE TABLE statement or ALTER TABLE statement.
- Set the **IFX_DEF_TABLE_LOCKMODE** environment variable to ROW so that all tables you subsequently create within a session use ROW without the need to specify the lock mode in the CREATE TABLE statement or ALTER TABLE statement.
- Set the DEF_TABLE_LOCKMODE configuration parameter to ROW so that all tables subsequently created within the database server use ROW without the need to specify the lock mode in the CREATE TABLE statement or ALTER TABLE statement.

If you change the lock mode with the **IFX_DEF_TABLE_LOCKMODE** environment variable or DEF_TABLE_LOCKMODE configuration parameter, the lock mode of existing tables are not affected. Existing tables continue to use the lock mode with which they were defined at the time they were created.

In addition, if you previously changed the lock mode of a table to ROW, and subsequently execute an ALTER TABLE statement to alter some other characteristic of the table (such as add a column or change the extent size), you do not need to specify the lock mode. The lock mode remains at ROW and is not set to the default PAGE mode.

You can still override the lock mode of individual tables by specifying the LOCK MODE clause in the CREATE TABLE statement or ALTER TABLE statement. The following list shows the order of precedence for the lock mode on a table:

- The system default is page locks. The database server uses this system default if you do not set the configuration parameter, do not set the environment variable, or do not specify the LOCK MODE clause in the SQL statements.
- If you set the DEF_TABLE_LOCKMODE configuration parameter, the database server uses this value when you do not set the environment variable, or do not specify the LOCK MODE clause in the SQL statements.
- If you set the **IFX_DEF_TABLE_LOCKMODE** environment variable, this value overrides the DEF_TABLE_LOCKMODE configuration parameter and system default. The database server uses this value when you do not specify the LOCK MODE clause in the SQL statements.
- If you specify the LOCK MODE clause in the CREATE TABLE statement or ALTER TABLE statement, this value overrides the **IFX_DEF_TABLE_LOCKMODE,** the DEF_TABLE_LOCKMODE configuration parameter and system default.

Lock Waits

When a user encounters a lock, the default behavior of a database server is to return an error to the application. You can execute the following SQL statement to wait indefinitely for a lock:

```
SET LOCK MODE TO WAIT;
```

You can also wait for a specific number of seconds, as in the following example:

```
SET LOCK MODE TO WAIT 20;
```

To return to the default behavior (no waiting for locks), execute the following statement:

```
SET LOCK MODE TO NOT WAIT;
```

Locks with the SELECT Statement

The type and duration of locks that the database server places depend on the isolation level set in the application, the database mode (logging, nonlogging, or ANSI,) and on whether the SELECT statement is within an update cursor. These locks can affect overall performance because they affect concurrency, as the following sections describe.

Isolation Level

The number and duration of locks placed on data during a SELECT statement depend on the level of isolation that the user sets. The type of isolation can affect overall performance because it affects concurrency.

You can set the isolation level with the SET ISOLATION or the ANSI SET TRANSACTION statement before you execute the SELECT statement. The main differences between the two statements are that SET ISOLATION has an additional isolation level, Cursor Stability, and SET TRANSACTION cannot be executed more than once in a transaction as SET ISOLATION can.

Dirty Read Isolation

Dirty Read isolation (or ANSI Read Uncommitted) places no locks on any rows fetched during a SELECT statement. Dirty Read isolation is appropriate for static tables that are used for queries.

Use Dirty Read with care if update activity occurs at the same time. With Dirty Read, the reader can read a row that has not been committed to the database and might be eliminated or changed during a rollback. For example, consider the following scenario:

```
User 1 starts a transacion.
User 1 inserts row A.
User 2 reads row A.
User 1 rolls back row A.
```

User 2 reads row A, which user 1 rolls back seconds later. In effect, user 2 read a row that was never committed to the database. Sometimes known as a *phantom row*, uncommitted data that is rolled back can pose a problem for applications.

Because the database server does not check or place any locks for queries, Dirty Read isolation offers the best performance of all isolation levels. However, because of potential problems with phantom rows, use it with care.

Because phantom rows are an issue only with transactions, databases that do not have logging on (and hence do not allow transactions) use Dirty Read as a default isolation level. In fact, Dirty Read is the only isolation level allowed for databases without logging on.

Committed Read Isolation

Committed Read isolation (or ANSI Read Committed) removes the problem of phantom reads. A reader with this isolation level checks for locks before it returns a row. By checking for locks, the reader cannot return any uncommitted rows.

The database server does not actually place any locks for rows read during Committed Read. It simply checks for any existing rows in the internal lock table.

Committed Read is the default isolation level for databases with logging, and it is an appropriate isolation level for most activities.

Cursor Stability Isolation

A reader with Cursor Stability isolation acquires a shared lock on the row that is currently fetched. This action assures that no other user can update the row until the user fetches a new row.

The pseudocode in Figure 8-1 shows when the database server places and releases locks with a cursor.

If you do not use a cursor to fetch data, Cursor Stability isolation behaves in the same way as Committed Read. No locks are actually placed.

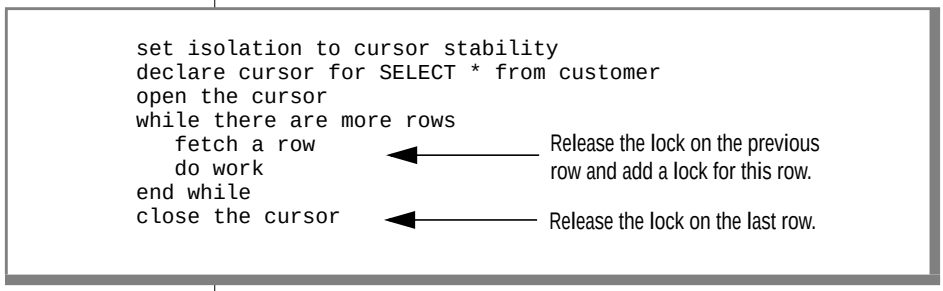

set isolation to cursor stability
declare cursor for SELECT * from customer
open the cursor
while there are more rows
fetch a row
do work
end while
close the cursor
Release the lock on the previous row and add a lock for this row.
Release the lock on the last row.

Figure 8-1
Locks Placed for Cursor Stability

Repeatable Read Isolation

Repeatable Read isolation (ANSI Serializable and ANSI Repeatable Read) is the strictest isolation level. With Repeatable Read, the database server locks all rows examined (not just fetched) for the duration of the transaction.

The pseudocode in Figure 8-2 shows when the database server places and releases locks with a cursor.

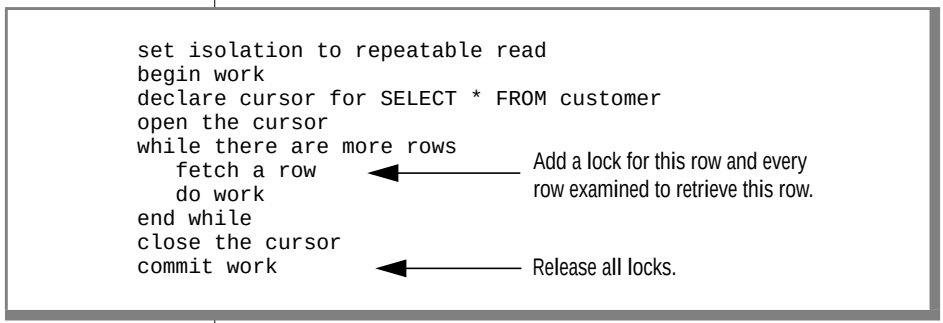

set isolation to repeatable read
begin work
declare cursor for SELECT * FROM customer
open the cursor
while there are more rows
fetch a row
do work
end while
close the cursor
commit work
Add a lock for this row and every row examined to retrieve this row.
Release all locks.

Figure 8-2
Locks Placed for Repeatable Read

Repeatable Read is useful during any processing in which multiple rows are examined, but none must change during the transaction. For example, suppose an application must check the account balance of three accounts that belong to one person. The application gets the balance of the first account and then the second. But, at the same time, another application begins a transaction that debits the third account and credits the first account. By the time that the original application obtains the account balance of the third account, it has been debited. However, the original application did not record the debit of the first account.

When you use Committed Read or Cursor Stability, the previous scenario can occur. However, it cannot occur with Repeatable Read. The original application holds a read lock on each account that it examines until the end of the transaction, so the attempt by the second application to change the first account fails (or waits, depending upon SET LOCK MODE).

Because even examined rows are locked, if the database server reads the table sequentially, a large number of rows unrelated to the query result can be locked. For this reason, use Repeatable Read isolation for tables when the database server can use an index to access a table. If an index exists and the optimizer chooses a sequential scan instead, you can use directives to force use of the index. However, forcing a change in the query path might negatively affect query performance.

Locking Nonlogging Tables

The database server does not place page or row locks on a nonlogging table when you use the table within a transaction. Use one of the following methods to prevent concurrency problems when other users are modifying a nonlogging table:

- Lock the table in exclusive mode for the whole transaction.
- Use Repeatable Read isolation level for the whole transaction.

Important: *Nonlogging raw tables are intended for fast loading of data. Informix recommends that you change the table to STANDARD before you use it in a transaction or modify the data within it.*

Update Cursors

An update cursor is a special kind of cursor that applications can use when the row might potentially be updated. To use an update cursor, execute SELECT FOR UPDATE in your application. Update cursors use *promotable locks*; that is, the database server places an update lock (meaning other users can still view the row) on the row when the application fetches the row, but the lock is changed to an exclusive lock when the application uses an update cursor and UPDATE...WHERE CURRENT OF to update the row.

In some cases, the database server might place locks on rows that the database server has examined but not actually fetched. Whether this behavior occurs depends on how the database server executes the SQL statement.

The advantage of an update cursor is that you can view the row with the confidence that other users cannot change it or view it with an update cursor while you are viewing it and before you update it.

If you do not update the row, the default behavior of the database server is to release the update lock when you execute the next FETCH statement or close the cursor. However, if you execute the SET ISOLATION statement with the RETAIN UPDATE LOCKS clause, the database server does not release any currently existing or subsequently placed update locks until the end of the transaction.

The pseudocode in Figure 8-3 shows when the database server places and releases update locks with a cursor. The database server releases the update lock on row one as soon as the next fetch occurs. However, after the database server executes the SET ISOLATION statement with the RETAIN UPDATE LOCKS clause, it does not release any update locks until the end of the transaction.

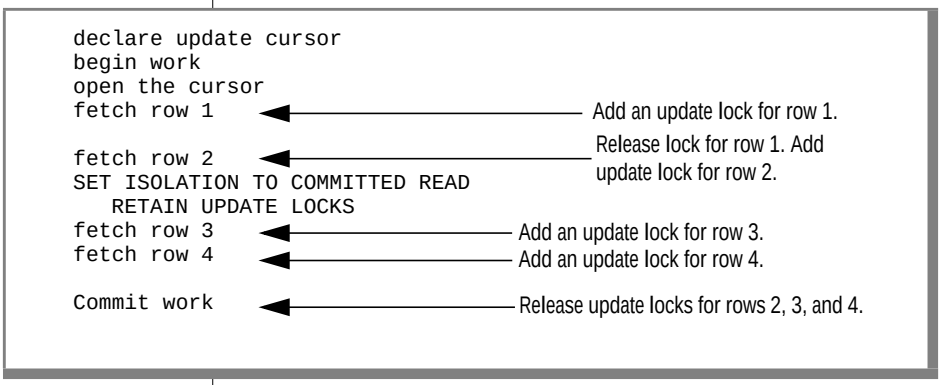

declare update cursor
begin work
open the cursor
fetch row 1
Add an update lock for row 1.
fetch row 2
Release lock for row 1. Add update lock for row 2.
SET ISOLATION TO COMMITTED READ
RETAIN UPDATE LOCKS
fetch row 3
Add an update lock for row 3.
fetch row 4
Add an update lock for row 4.
Commit work
Release update locks for rows 2, 3, and 4.

Figure 8-3
When Update Locks Are Released

In an ANSI-compliant database, update cursors are usually not needed because any select cursor behaves the same as an update cursor without the RETAIN UPDATE LOCKS clause.

Figure 8-4 shows the database server promoting an update lock to an exclusive lock.

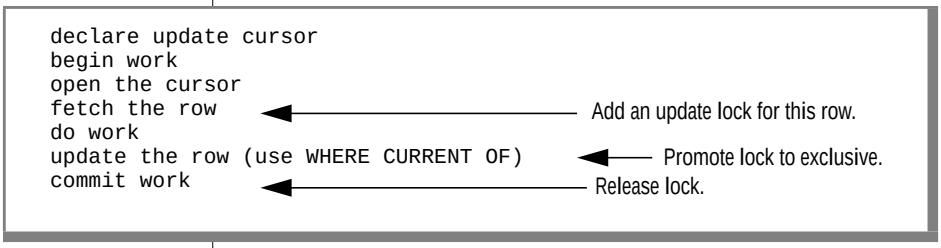


Figure 8-4
When Update Locks Are Promoted

Locks Placed with INSERT, UPDATE, and DELETE

When you execute an INSERT, UPDATE, or DELETE statement, the database server uses exclusive locks. An exclusive lock means that no other users can view the row unless they are using the Dirty Read isolation level. In addition, no other users can update or delete the item until the database server removes the lock.

When the database server removes the exclusive lock depends on the type of logging set for the database. If the database has logging, the database server removes all exclusive locks when the transaction completes (commits or rolls back). If the database does not have logging, the database server removes all exclusive locks immediately after the INSERT, UPDATE, or DELETE statement completes.

Monitoring and Administering Locks

The database server stores locks in an internal lock table. When the database server reads a row, it checks if the row or its associated page, table, or database is listed in the lock table. If it is in the lock table, the database server must also check the lock type. The following table shows the types of locks that the lock table can contain.

Lock Type	Description	Statement That Usually Places the Lock
S	Shared lock	SELECT
X	Exclusive lock	INSERT, UPDATE, DELETE
U	Update lock	SELECT in an update cursor
B	Byte lock	Any statement that updates VARCHAR columns

A byte lock is generated only if you shrink the size of a data value in a VARCHAR column. The byte lock exists solely for roll forward and rollback execution, so a byte lock is created only if you are working in a database that uses logging. Byte locks appear in **onstat -k** output only if you are using row-level locking; otherwise, they are merged with the page lock.

In addition, the lock table might store *intent locks*, with the same lock type as previously shown. In some cases, a user might need to register his or her possible intent to lock an item, so that other users cannot place a lock on the item.

Depending on the type of operation and the isolation level, the database server might continue to read the row and place its own lock on the row, or it might wait for the lock to be released (if the user executed SET LOCK MODE TO WAIT). The following table shows the locks that a user can place if another user holds a certain type of lock. For example, if one user holds an exclusive lock on an item, another user requesting any kind of lock (exclusive, update, or shared) receives an error.

	Hold X lock	Hold U lock	Hold S lock
Request X lock	No	No	Yes
Request U lock	No	No	Yes
Request S lock	No	Yes	Yes

Monitoring Locks

You can view the lock table with **onstat -k**. Figure 8-5 shows sample output for **onstat -k**.

```
Locks
address  wtlist   owner    lklist   type     tblsnum  rowid    key#/bsiz
300b77d0 0        40074140 0        HDR+S    10002    106        0
300b7828 0        40074140 300b77d0 HDR+S    10197    123        0
300b7854 0        40074140 300b7828 HDR+IX   101e4    0          0
300b78d8 0        40074140 300b7854 HDR+X    101e4    102        0
 4 active, 5000 total, 8192 hash buckets
```

Figure 8-5
onstat -k Output

In this example, a user is inserting one row in a table. The user holds the following locks (described in the order shown):

- A shared lock on the database
- A shared lock on a row in the **systables** system catalog table
- An intent-exclusive lock on the table
- An exclusive lock on the row

To determine the table to which the lock applies, execute the following SQL statement. For *tblsnum*, substitute the value shown in the **tblsnum** field in the **onstat -k** output.

```
SELECT tabname
FROM systables
WHERE partnum = hex(tblsnum)
```

You can also query the **syslocks** table in the **systables** database to obtain information on each active lock. The **syslocks** table contains the following columns.

Column	Description
dbsname	Database on which the lock is held
tabname	Name of the table on which the lock is held
rowidlk	ID of the row on which the lock is held (`0` indicates table lock)
keynum	The key number for the row
type	Type of lock
owner	Session ID of the lock owner
waiter	Session ID of the first waiter on the lock

Configuring and Monitoring the Number of Locks

The LOCKS configuration parameter sets the initial size of the internal lock table. If the number of locks allocated by sessions exceeds the value of LOCKS, the database server increases the lock table by doubling its size. Each time that the lock table overflows (when it is greater than the size of LOCKS), the database server doubles the size of the lock table, up to 15 times. Each time that the database server doubles the size of the lock table, it allocates no more than 100,000 locks. After the fifteenth time that the database server doubles the lock table, it no longer increases the size of the lock table, and an application needing a lock receives an error. For more information on how to determine an initial value for the LOCKS configuration parameter, refer to "LOCKS" on page 4-21.

To monitor the number of times that applications receive the out-of-locks error, view the **ovlock** field in the output of **onstat -p**. You can also see similar information from the **sysprofile** table in the sysmaster database. The following rows contain the relevant statistics.

Row	Description
ovlock	Number of times that sessions attempted to exceed the maximum number of locks
lockreqs	Number of times that sessions requested a lock
lockwts	Number of times that sessions had to wait for a lock

Every time the database server increases the size of the lock table, it places a message in the message log file. You should monitor the message-log file periodically and increase the size of the LOCK parameter if you see that the database server has increased the size of the lock table.

The lock table can hold up to 9,500,000 locks, which is the maximum value for the LOCKS parameter (8,000,000) plus 15 dynamic allocations of 100,000 locks each. However, a very large lock table can impede performance. Although the algorithm to read the lock table is efficient, you incur some cost for reading a very large table each time that the database server reads a row. If the database server is using an unusually large number of locks, you might need to examine how individual applications are using locks.

First, monitor sessions with **onstat -u** to see if a particular user is using an especially high number of locks (a high value in the **locks** column). If a particular user uses a large number of locks, examine the SQL statements in the application to determine whether you should lock the table or use individual row or page locks. A table lock is more efficient than individual row locks, but it reduces concurrency.

One way to reduce the number of locks placed on a table is to alter a table to use page locks instead of row locks. However, page locks reduce overall concurrency for the table, which can affect performance.

Monitoring Lock Waits and Lock Errors

If the application executes SET LOCK MODE TO WAIT, the database server waits for a lock to be released instead of returning an error. An unusually long wait for a lock can give users the impression that the application is hanging.

In Figure 8-6, the **onstat -u** output shows that session ID 84 is waiting for a lock (L in the first column of the **Flags** field). To find out the owner of the lock, use the **onstat -k** command.

```
onstat -u

Userthreads
address  flags   sessid user     tty    wait    tout locks nreads  nwrites
40072010 ---P--D 7      informix -        0       0     0     35       75
400723c0 ---P--- 0      informix -        0       0     0     0        0
40072770 ---P--- 1      informix -        0       0     0     0        0
40072b20 ---P--- 2      informix -        0       0     0     0        0
40072ed0 ---P--F 0      informix -        0       0     0     0        0
40073280 ---P--B 8      informix -        0       0     0     0        0
40073630 ---P--- 9      informix -        0       0     0     0        0
400739e0 ---P--D 0      informix -        0       0     0     0        0
40073d90 ---P--- 0      informix -        0       0     0     0        0
40074140 Y-BP--- 81        lsuto   4 50205788     0     4   106      221
400744f0 --BP--- 83        jsmit   -      0       0     4     0        0
400753b0 ---P--- 86       worth     -      0       0     2     0        0
40075760 L--PR-- 84        jones   3 300b78d8    -1     2     0        0
 13 active, 128 total, 16 maximum concurrent

onstat -k

Locks
address  wtlist   owner    lklist   type     tblsnum  rowid    key#/bsiz
300b77d0 0        40074140 0        HDR+S    10002    106        0
300b7828 0        40074140 300b77d0 HDR+S    10197    122        0
300b7854 0        40074140 300b7828 HDR+IX   101e4    0          0
300b78d8 40075760 40074140 300b7854 HDR+X    101e4    100        0
300b7904 0        40075760 0              S  10002    106        0
300b7930 0        40075760 300b7904     S    10197    122        0
 6 active, 5000 total, 8192 hash buckets
```

Figure 8-6
onstat -u Output That Shows Lock Usage

To find out the owner of the lock for which session ID 84 is waiting

1. Obtain the address of the lock in the **wait** field (`300b78d8`) of the **onstat -u** output.
2. Find this address (`300b78d8`) in the **Locks address** field of the **onstat -k** output.

   The **owner** field of this row in the **onstat -k** output contains the address of the user thread (`40074140`).
3. Find this address (`40074140`) in the **Userthreads** field of the **onstat -u** output.

   The **sessid** field of this row in the **onstat -u** output contains the session ID (`81`) that owns the lock.

To eliminate the contention problem, you can have the user exit the application gracefully. If this solution is not possible, you can kill the application process or remove the session with **onmode -z**.

Monitoring Deadlocks

A *deadlock* occurs when two users hold locks, and each user wants to acquire a lock that the other user owns.

For example, user **joe** holds a lock on row 10. User **jane** holds a lock on row 20. Suppose that **jane** wants to place a lock on row 10, and **joe** wants to place a lock on row 20. If both users execute SET LOCK MODE TO WAIT, they potentially might wait for each other forever.

Informix uses the lock table to detect deadlocks automatically and stop them before they occur. Before a lock is granted, the database server examines the lock list for each user. If a user holds a lock on the resource that the requestor wants to lock, the database server traverses the lock wait list for the user to see if the user is waiting for any locks that the requestor holds. If so, the requestor receives a deadlock error.

Deadlock errors can be unavoidable when applications update the same rows frequently. However, certain applications might always be in contention with each other. Examine applications that are producing a large number of deadlocks and try to run them at different times. To monitor the number of deadlocks, use the **deadlks** field in the output of **onstat -p**.

In a distributed transaction, the database server does not examine lock tables from other database server systems, so deadlocks cannot be detected before they occur. Instead, you can set the DEADLOCK_TIMEOUT parameter. DEADLOCK_TIMEOUT specifies the number of seconds that the database server waits for a remote database server response before it returns an error. Although reasons other than a distributed deadlock might cause the delay, this mechanism keeps a transaction from hanging indefinitely.

To monitor the number of distributed deadlock timeouts, use the **dltouts** field in the **onstat -p** output.

Monitoring Isolation That Sessions Use

The **onstat -g** ses and **onstat -g sql** statements list the isolation level that a session is currently using. The following table summarizes the values in the **IsoLvl** column.

Value	Description
DR	Dirty Read
CR	Committed Read
CS	Cursor Stability
RR	Repeatable Read
DRU	Dirty Read with RETAIN UPDATE LOCKS
CRU	Committed Read with RETAIN UPDATE LOCKS
CSU	Cursor Stability with RETAIN UPDATE LOCKS

If a great deal of lock contention occurs, check the isolation level of sessions to make sure it is appropriate for the application.

Locks for Smart Large Objects

Smart large objects have several unique locking behaviors because their columns are typically much larger than other columns in a table. This section discusses the following unique behaviors:

- Types of locks on smart large objects
- Byte-range locking
- Lock promotion
- Dirty Read isolation level with smart large objects

Types of Locks on Smart Large Objects

The database server uses one of the following granularity levels for locking smart large objects:

- The sbspace chunk header partition
- The smart large object
- A byte range of the smart large object

The default locking granularity is at the level of the smart large object. In other words, when you update a smart large object, by default the database server locks the smart large object that is being updated.

Locks on the sbspace chunk header partition only occur when the database server promotes locks on smart large objects. For more information, refer to "Lock Promotion" on page 8-27.

Byte-Range Locking

Rather than locking the entire smart large object, you can lock only a specific byte range of a smart large object. Byte-range locking is advantageous because it allows multiple users to update the same smart large object simultaneously, as long as they are updating different parts of it. Also, users can read a part of a smart large object while another user is updating or reading a different part of the same smart large object.

Figure 8-7 shows two locks placed on a single smart large object. The first lock is on bytes 2, 3, and 4. The second lock is on byte 6.

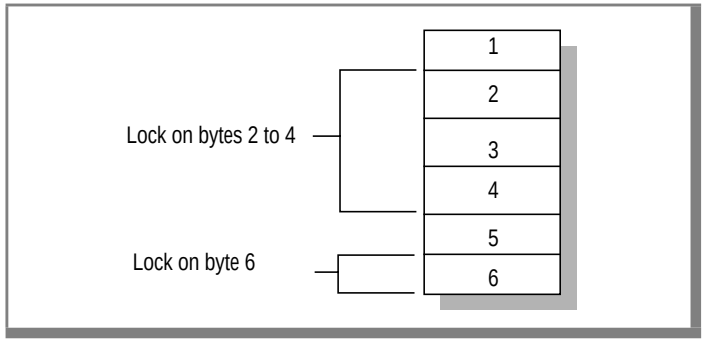


Figure 8-7
Example of Byte-Range Locking

How the Database Server Manages Byte-Range Locks

The database server manages byte-range locks in the lock table in a similar fashion to other locks placed on rows, pages, and tables. However, the lock table must store the byte range as well.

If a user places a second lock on a byte range adjacent to a byte range that the user currently has locked, the database server consolidates the two locks into one lock on the entire range. If a user holds locks that Figure 8-7 shows, and the user requests a lock on byte five, the database server consolidates the locks placed on bytes two through six into one lock.

Likewise, if a user unlocks only a portion of the bytes included within a byte-range lock, the database server might be split into multiple byte-range locks. In Figure 8-7, the user could unlock byte three, which causes the database server to change the one lock on bytes two through four to one lock on byte two and one lock on byte four.

Using Byte-Range Locks

By default, the database server places a lock on the smart large object. To use byte-range locks, you must perform one of the following actions:

- To set byte-range locking for the sbspace that stores the smart large object, use the **onspaces** utility. The following example sets byte-range locking for the new sbspace:

  ```
  onspaces -c -S slo -g 2 -p /ix/9.2/liz/slo -o 0 -s 1000
     -Df LOCK_MODE=RANGE
  ```

  When you set the default locking mode for the sbspace to byte-range locking, the database server locks only the necessary bytes when it updates any smart large objects stored in the sbspace.

- To set byte-range locking for the smart large object when you open it, use one of the following methods:

  DB API

  - Set the MI_LO_LOCKRANGE flag in the **mi_lo_open()** DataBlade API function. ♦

  E/C

  - Set the LO_LOCKRANGE flag in the **ifx_lo_open()** ESQL/C function. ♦

  When you set byte-range locking for the individual smart large object, the database server implicitly locks only the necessary bytes when it selects or updates the smart large object.

DB API

E/C

- To lock a byte range explicitly, use one of the following functions:
  - **mi_lo_lock()** ♦
  - **ifx_lo_lock()** ♦

  These functions lock the range of bytes that you specify for the smart large object. If you specify an exclusive lock with either function, UPDATE statements do not place locks on the smart large object if they update the locked bytes.

  The database server releases exclusive byte-range locks placed with **mi_lo_lock()** or **ifx_lo_lock()** at the end of the transaction. The database server releases shared byte-range locks placed with **mi_lo_lock()** or **ifx_lo_lock()** based on the same rules as locks placed with SELECT statements, depending upon the isolation level. You can also release shared byte-range locks with one of the following functions:

DB API

E/C

  - **mi_lo_unlock()** ♦
  - **ifx_lo_unlock()** ♦

For more information about the DataBlade API functions, refer to the *DataBlade API Programmer's Manual*. For more information about ESQL/C functions, refer to the *Informix ESQL/C Programmer's Manual*.

Monitoring Byte-Range Locks

Use **onstat -k** to list all byte-range locks. Use the **onstat -K** command to list byte-range locks and all waiters for byte-range locks. Figure 8-8 shows an excerpt from the output of **onstat -k**.

```
Byte-Range Locks
rowid/LOid   tblsnum  address  status   owner    offset   size     type
104          200004   a020e90  HDR
[2, 2, 3]             a020ee4  HOLD     a1b46d0  50       10           S
202          200004   a021034  HDR
[2, 2, 5]             a021088  HOLD     a1b51e0  40       5            S
102          200004   a035608  HDR
[2, 2, 1]             a0358fc  HOLD     a1b4148  0        500          S
                      a035758  HOLD     a1b3638  300      100          S
 21 active, 2000 total, 2048 hash buckets
```

Figure 8-8
Byte-Range Locks in onstat -k Output

Byte-range locks produce the following information in the **onstat -k** output.

Column	Description
rowid	The rowid of the row that contains the locked smart large object
LOid	The three values: sbspace number, chunk number, and sequence number (a value that represents the position in the chunk)
tblsnum	The number of the tblspace that holds the smart large object
address	The lock address
status	Status of the lock HDR is a placeholder. HOLD indicates the user specified in the **owner** column owns the lock. WAIT (shown only with **onstat -K**) indicates that the user specified in the owner column is waiting for the lock.
owner	The address of the owner (or waiter) Cross reference this value with the address in **onstat -u**.
offset	The offset into the smart large object where the bytes are locked
size	The number of bytes locked, starting at the value in the offset column
type	S (shared lock) or X (exclusive)

Setting Number of Locks for Byte-Range Locking

When you use byte-range locking, the database server can use more locks because of the possibility of multiple locks on one smart large object. Monitor the number of locks used with **onstat -k**. Even though the lock table grows when it runs out of space, you might want to increase the LOCKS parameter to match lock usage so that the database server does not have to allocate more space dynamically.

Lock Promotion

The database server uses lock promotion to decrease the total number of locks held on smart large objects. Too many locks can result in poorer performance because the database server frequently searches the lock table to determine if a lock exists on an object.

If the number of locks held by a transaction exceeds 33 percent of the current number of allocated locks for the database server, the database server attempts to promote any existing byte-range locks to a single lock on the smart large object.

If the number of locks that a user holds on a smart large objects (not on byte ranges of smart large objects) equals or exceeds 10 percent of the current capacity of the lock table, the database server attempts to promote all of the smart-large-object locks to one lock on the smart-large-object header partition. This kind of lock promotion improves performance for applications that are updating, loading, or deleting a large number of smart large objects. For example, a transaction that deletes millions of smart large objects would consume the entire lock table if the database server did not use lock promotion. The lock promotion algorithm has deadlock avoidance built in.

You can identify a smart-large-object header partition in **onstat -k** by `0` in the **rowid** column and a tablespace number with a high-order first byte-and-a-half that corresponds to the dbspace number where the smart large object is stored. For example, if the tblspace number is listed as `0x200004` (the high-order zeros are truncated), the dbspace number `2` corresponds to the dbspace number listed in **onstat -d**.

Even if the database server attempts to promote a lock, it might not be able to do so. For example, the database server might not be able to promote byte-range locks to one smart-large-object lock because other users have byte-range locks on the same smart large object. If the database server cannot promote a byte-range lock, it does not change the lock, and processing continues as normal.

Dirty Read and Smart Large Objects

You can use the Dirty Read isolation level for smart large objects. For information on how Dirty Reads affects consistency, refer to "Dirty Read Isolation" on page 8-10.

Set the Dirty Read isolation level for smart large objects in one of the following ways:

- Use the SET TRANSACTION MODE or SET ISOLATION statement.
- Use the LO_DIRTY_READ flag in one of the following functions:
  - **mi_lo_open()** ♦
  - **ifx_lo_open()** ♦

DB API

E/C

If consistency for smart large objects is not important, but consistency for other columns in the row is important, you can set the isolation level to Committed Read, Cursor Stability, or Repeatable Read and open the smart large object with the LO_DIRTY_READ flag.

Chapter

9

Fragmentation Guidelines

In This Chapter 9-3

Planning a Fragmentation Strategy 9-3
Setting Fragmentation Goals 9-4
Improving Performance for Individual Queries 9-5
Reducing Contention Between Queries and Transactions . . . 9-6
Increasing Data Availability 9-7
Increasing Granularity for Backup and Restore 9-8
Examining Your Data and Queries 9-8
Considering Physical Fragmentation Factors 9-9

Designing a Distribution Scheme 9-11
Choosing a Distribution Scheme 9-12
Designing an Expression-Based Distribution Scheme 9-14
Suggestions for Improving Fragmentation 9-15

Fragmenting Indexes 9-17
Attached Indexes 9-17
Detached Indexes 9-18
Restrictions on Indexes for Fragmented Tables 9-20

Fragmenting Temporary Tables 9-20

Using Distribution Schemes to Eliminate Fragments 9-21
Fragmentation Expressions for Fragment Elimination 9-22
Query Expressions for Fragment Elimination 9-22
Range Expressions in Query 9-23
Equality Expressions in Query 9-24
Effectiveness of Fragment Elimination 9-24
Nonoverlapping Fragments on a Single Column 9-26
Overlapping Fragments on a Single Column 9-27
Nonoverlapping Fragments, Multiple Columns 9-28

Improving the Performance of Attaching and Detaching Fragments . . 9-29
Improving ALTER FRAGMENT ATTACH Performance 9-29
Formulating Appropriate Distribution Schemes 9-30
Ensuring No Data Movement When You Attach a Fragment . . 9-33
Updating Statistics on All Participating Tables 9-34
Improving ALTER FRAGMENT DETACH Performance 9-37
Fragmenting the Index in the Same Way as the Table 9-37
Fragmenting the Index Using Same Distribution Scheme as the Table 9-38

Monitoring Fragment Use 9-39
Using the onstat Utility 9-39
Using SET EXPLAIN 9-40

In This Chapter

This chapter discusses the performance considerations that are involved when you use table fragmentation.

One of the most frequent causes of poor performance in relational database systems is contention for data that resides on a single I/O device. Informix database servers support table fragmentation (also *partitioning*), which allows you to store data from a single table on multiple disk devices. Proper fragmentation of high-use tables can significantly reduce I/O contention.

For information about fragmentation and parallel execution, refer to Chapter 12, "Parallel Database Query."

For an introduction to fragmentation concepts and methods, refer to the *Informix Guide to Database Design and Implementation*. For information about the SQL statements that manage fragments, refer to the *Informix Guide to SQL: Syntax*.

Planning a Fragmentation Strategy

A fragmentation strategy consists of two parts:

- A distribution scheme that specifies how to group rows into fragments

  You specify the distribution scheme in the FRAGMENT BY clause of the CREATE TABLE, CREATE INDEX, or ALTER FRAGMENT statements.

- The set of dbspaces (or dbslices) in which you locate the fragments

  You specify the set of dbspaces or dbslices in the IN clause (storage option) of these SQL statements.

To formulate a fragmentation strategy

1. Decide on your primary fragmentation goal, which should depend, to a large extent, on the types of applications that access the table.
2. Make the following decisions based on your primary fragmentation goal:
   - Whether to fragment the table data, the table index, or both
   - What the ideal distribution of rows or index keys is for the table
3. Choose between the following distribution schemes.
   - If you choose an expression-based distribution scheme, you must then design suitable fragment expressions.
   - If you choose a round-robin distribution scheme, the database server determines which rows to put into a specific fragment.
4. To complete the fragmentation strategy, you must decide on the number and location of the fragments:
   - The number of fragments depends on your primary fragmentation goal.
   - Where you locate fragments depends on the number of disks available in your configuration.

Setting Fragmentation Goals

Analyze your application and workload to determine the balance to strike among the following fragmentation goals:

- Improved performance for individual queries

  To improve the performance of individual queries, fragment tables appropriately and set resource-related parameters to specify system resource use (memory, CPU virtual processors, and so forth).
- Reduced contention between queries and transactions

- If your database server is used primarily for OLTP transactions and only incidentally for decision-support queries, you can often use fragmentation to reduce contention when simultaneous queries against a table perform index scans to return a few rows.Increased data availability

  Careful fragmentation of dbspaces can improve data availability if devices fail. Table fragments on the failed device can be restored quickly, and other fragments are still accessible.

- Improved data-load performance

  When you use the High-Performance Loader (HPL) to load a table that is fragmented across multiple disks, it allocates threads to light append the data into the fragments in parallel. For more information on this load method, refer to the *Guide to the High-Performance Loader*.

  You can also use the ALTER FRAGMENT ON TABLE statement with the ATTACH clause to add data quickly to a very large table. For more information, refer to "Improving the Performance of Attaching and Detaching Fragments" on page 9-29.

The performance of a fragmented table is primarily governed by the following factors:

- The storage option that you use for allocating disk space to fragments (discussed in "Considering Physical Fragmentation Factors" on page 9-9)
- The distribution scheme used to assign rows to individual fragments (discussed in "Designing a Distribution Scheme" on page 9-11)

Improving Performance for Individual Queries

If the primary goal of fragmentation is improved performance for individual queries, try to distribute all the rows of the table evenly over the different disks. Overall query-completion time is reduced when the database server does not have to wait for data retrieval from a table fragment that has more rows than other fragments.

If queries access data by performing sequential scans against significant portions of tables, fragment the table rows only. Do not fragment the index. If an index is fragmented and a query has to cross a fragment boundary to access the data, the performance of the query can be worse than if you do not fragment.

If queries access data by performing an index read, you can improve performance by using the same distribution scheme for the index and the table.

If you use round-robin fragmentation, do not fragment your index. Consider placing that index in a separate dbspace from other table fragments.

For more information about improving performance for queries, see "Query Expressions for Fragment Elimination" on page 9-22 and Chapter 13, "Improving Individual Query Performance."

Reducing Contention Between Queries and Transactions

Fragmentation can reduce contention for data in tables that multiple queries and OLTP applications use. Fragmentation often reduces contention when many simultaneous queries against a table perform index scans to return a few rows. For tables subjected to this type of load, fragment both the index keys and data rows with a distribution scheme that allows each query to eliminate unneeded fragments from its scan. Use an expression-based distribution scheme. For more information, refer to "Using Distribution Schemes to Eliminate Fragments" on page 9-21.

To fragment a table for reduced contention, start by investigating which queries access which parts of the table. Next, fragment your data so that some of the queries are routed to one fragment while others access a different fragment. The database server performs this routing when it evaluates the fragmentation rule for the table. Finally, store the fragments on separate disks.

Your success in reducing contention depends on how much you know about the distribution of data in the table and the scheduling of queries against the table. For example, if the distribution of queries against the table is set up so that all rows are accessed at roughly the same rate, try to distribute rows evenly across the fragments. However, if certain values are accessed at a higher rate than others, you can compensate for this difference by distributing the rows over the fragments to balance the access rate. For more information, refer to "Designing an Expression-Based Distribution Scheme" on page 9-14.

Increasing Data Availability

When you distribute table and index fragments across different disks or devices, you improve the availability of data during disk or device failures. The database server continues to allow access to fragments stored on disks or devices that remain operational. This availability has important implications for the following types of applications:

- Applications that do not require access to unavailable fragments

  A query that does not require the database server to access data in an unavailable fragment can still successfully retrieve data from fragments that are available. For example, if the distribution expression uses a single column, the database server can determine if a row is contained in a fragment without accessing the fragment. If the query accesses only rows that are contained in available fragments, a query can succeed even when some of the data in the table is unavailable. For more information, refer to "Designing an Expression-Based Distribution Scheme" on page 9-14.

- Applications that accept the unavailability of data

  Some applications might be designed in such a way that they can accept the unavailability of data in a fragment and require the ability to retrieve the data that is available. To specify which fragments can be skipped, these applications can execute the SET DATASKIP statement before they execute a query. Alternatively, the database server administrator can use the onspaces -f option to specify which fragments are unavailable.

If your fragmentation goal is increased availability of data, fragment both table rows and index keys so that if a disk drive fails, some of the data is still available. If applications must always be able to access a subset of your data, keep those rows together in the same mirrored dbspace.

Increasing Granularity for Backup and Restore

Consider the following two backup and restore factors when you are deciding how to distribute dbspaces across disks:

- **Data availability.** When you decide where to place your tables or fragments, remember that if a device that contains a dbspace fails, all tables or table fragments in that dbspace are inaccessible, even though tables and fragments in other dbspaces are accessible. The need to limit data unavailability in the event of a disk failure might influence which tables you group together in a particular dbspace.
- **Cold versus warm restores.** Although you must perform a cold restore if a dbspace that contains critical data fails, you need to perform only a warm restore if a noncritical dbspace fails. The desire to minimize the impact of cold restores might influence the dbspace that you use to store critical data.

For more information about backup and restore, see your *Backup and Restore Guide* or *Archive and Backup Guide*.

Examining Your Data and Queries

To determine a fragmentation strategy, you must know how the data in a table is used. Take the following steps to gather information about a table that you might fragment.

To gather information about your table

1. Identify the queries that are critical to performance to determine if the queries are OLTP or DSS.
2. Use the SET EXPLAIN statement to determine how the data is being accessed.

   For information on the output of the SET EXPLAIN statement, refer to "Query Plan Report" on page 10-12. To determine how the data is accessed, you can sometimes simply review the SELECT statements along with the table schema.
3. Determine what portion of the data each query examines.

   For example, if certain rows in the table are read most of the time, you can isolate them in a small fragment to reduce I/O contention for other fragments.

4. Determine which statements create temporary files.

   Decision-support queries typically create and access large temporary files, and placement of temporary dbspaces can be critical to performance.

5. If particular tables are always joined together in a decision-support query, spread fragments for these tables across different disks.

6. Examine the columns in the table to determine which fragmentation scheme would keep each scan thread equally busy for the decision-support queries.

   To see how the column values are distributed, create a distribution on the column with the UPDATE STATISTICS statement and examine the distribution with **dbschema**.

   ```
   dbschema -d database -hd table
   ```

Considering Physical Fragmentation Factors

When you fragment a table, the physical placement issues that pertain to tables apply to individual table fragments. For details, refer to Chapter 6, "Table Performance Considerations." Because each fragment resides in its own dbspace on a disk, you must address these issues separately for the fragments on each disk.

Fragmented and nonfragmented tables differ in the following ways:

- For fragmented tables, each fragment is placed in a separate, designated dbspace. For nonfragmented tables, the table can be placed in the default dbspace of the current database. Regardless of whether the table is fragmented or not, Informix recommends that you create a single chunk on each disk for each dbspace.
- Extent sizes for a fragmented table are usually smaller than the extent sizes for an equivalent nonfragmented table because fragments do not grow in increments as large as the entire table. For more information on how to estimate the space to allocate, refer to "Estimating Table Size" on page 6-12.

- In a fragmented table, the row pointer is not a unique unchanging pointer to the row on a disk. The database server uses the combination of fragment ID and row pointer internally, inside an index, to point to the row. These two fields are unique but can change over the life of the row. An application cannot access the fragment ID; therefore, Informix recommends that you use primary keys to access a specific row in a fragmented table. For more information, refer to the *Informix Guide to Database Design and Implementation*.
- An attached index or an index on a nonfragmented table uses 4 bytes for the row pointer. A detached index uses 8 bytes of disk space per key value for the fragment ID and row pointer combination. For more information on how to estimate space for an index, refer to "Estimating lndex Pages" on page 7-3. For more information on attached indexes and detached indexes, refer to "Fragmenting Indexes" on page 9-17.

Decision-support queries usually create and access large temporary files; placement of temporary dbspaces is a critical factor for performance. For more information about placement of temporary files, refer to "Spreading Temporary Tables and Sort Files Across Multiple Disks" on page 6-10.

Designing a Distribution Scheme

After you decide whether to fragment table rows, index keys, or both, and you decide how the rows or keys should be distributed over fragments, you decide on a scheme to implement this distribution.

The database server supports the following distribution schemes:

- **Round-robin.** This type of fragmentation places rows one after another in fragments, rotating through the series of fragments to distribute the rows evenly.

  For smart large objects, you can specify multiple sbspaces in the PUT clause of the CREATE TABLE or ALTER TABLE statement to distribute smart large objects in a round-robin distribution scheme so that the number of smart large objects in each space is approximately equal.

  For INSERT statements, the database server uses a hash function on a random number to determine the fragment in which to place the row. For INSERT cursors, the database server places the first row in a random fragment, the second in the next fragment sequentially, and so on. If one of the fragments is full, it is skipped.

- **Expression-based.** This type of fragmentation puts rows that contain specified values in the same fragment. You specify a *fragmentation expression* that defines criteria for assigning a set of rows to each fragment, either as a range rule or some arbitrary rule. You can specify a *remainder fragment* that holds all rows that do not match the criteria for any other fragment, although a remainder fragment reduces the efficiency of the expression-based distribution scheme.

Choosing a Distribution Scheme

Figure 9-1 compares round-robin and expression-based distribution schemes for three important features.

Figure 9-1
Distribution-Scheme Comparisons

Distribution Scheme	Ease of Data Balancing	Fragment Elimination	Data Skip
Round-robin	Automatic. Data is balanced over time.	The database server cannot eliminate fragments.	You cannot determine if the integrity of the transaction is compromised when you use the data-skip feature. However, you can insert into a table fragmented by round-robin.
Expression-based	Requires knowledge of the data distribution.	If expressions on one or two columns are used, the database server can eliminate fragments for queries that have either range or equality expressions.	You can determine whether the integrity of a transaction has been compromised when you use the data-skip feature. You cannot insert rows if the appropriate fragment for those rows is down.

The distribution scheme that you choose depends on the following factors:

- The features in Figure 9-1 of which you want to take advantage
- Whether or not your queries tend to scan the entire table
- Whether or not you know the distribution of data to be added
- Whether or not your applications tend to delete many rows
- Whether or not you cycle your data through the table

Basically, the round-robin scheme provides the easiest and surest way of balancing data. However, with round-robin distribution, you have no information about the fragment in which a row is located, and the database server cannot eliminate fragments.

In general, round-robin is the correct choice only when all the following conditions apply:

- Your queries tend to scan the entire table.
- You do not know the distribution of data to be added.
- Your applications tend not to delete many rows. (If they do, load balancing could be degraded.)

An expression-based scheme might be the best choice to fragment the data if any of the following conditions apply:

- Your application calls for numerous decision-support queries that scan specific portions of the table.
- You know what the data distribution is.
- You plan to cycle data through a database.

If you plan to add and delete large amounts of data periodically, based on the value of a column such as date, you can use that column in the distribution scheme. You can then use the ALTER FRAGMENT ATTACH and ALTER FRAGMENT DETACH statements to cycle the data through the table.

The ALTER FRAGMENT ATTACH and DETACH statements provide the following advantages over bulk loads and deletes:

- The rest of the table fragments are available for other users to access. Only the fragment that you attach or detach is not available to other users.
- With the performance enhancements, the execution of an ALTER FRAGMENT ATTACH or DETACH statement is much faster than a bulk load or mass delete.

For more information, refer to "Improving the Performance of Attaching and Detaching Fragments" on page 9-29.

In some cases, an appropriate index scheme can circumvent the performance problems of a particular distribution scheme. For more information, refer to "Fragmenting Indexes" on page 9-17.

Designing an Expression-Based Distribution Scheme

The first step in designing an expression-based distribution scheme is to determine the distribution of data in the table, particularly the distribution of values for the column on which you base the fragmentation expression. To obtain this information, run the UPDATE STATISTICS statement for the table and then use the **dbschema** utility to examine the distribution.

Once you know the data distribution, you can design a fragmentation rule that distributes data across fragments as required to meet your fragmentation goal. If your primary goal is to improve performance, your fragment expression should generate an even distribution of rows across fragments.

If your primary fragmentation goal is improved concurrency, analyze the queries that access the table. If certain rows are accessed at a higher rate than others, you can compensate by opting for an uneven distribution of data over the fragments that you create.

Try not to use columns that are subject to frequent updates in the distribution expression. Such updates can cause rows to move from one fragment to another (that is, be deleted from one and added to another), and this activity increases CPU and I/O overhead.

Try to create nonoverlapping regions based on a single column with no REMAINDER fragment for the best fragment-elimination characteristics. The database server eliminates fragments from query plans whenever the query optimizer can determine that the values selected by the WHERE clause do not reside on those fragments, based on the expression-based fragmentation rule by which you assign rows to fragments. For more information, refer to "Using Distribution Schemes to Eliminate Fragments" on page 9-21.

Suggestions for Improving Fragmentation

The following suggestions are guidelines for fragmenting tables and indexes:

- For optimal performance in decision-support queries, fragment the table to increase parallelism, but do not fragment the indexes. Detach the indexes, and place them in a separate dbspace.
- For best performance in OLTP, use fragmented indexes to reduce contention between sessions. You can often fragment an index by its key value, which means the OLTP query only has to look at one fragment to find the location of the row.

  If the key value does not reduce contention, as when every user looks at the same set of key values (for instance, a date range), consider fragmenting the index on another value used in the WHERE clause. To cut down on fragment administration, consider not fragmenting some indexes, especially if you cannot find a good fragmentation expression to reduce contention.
- Use round-robin fragmentation on data when the table is read sequentially by decision-support queries. Round-robin fragmentation is a good method for spreading data evenly across disks when no column in the table can be used for an expression-based fragmentation scheme. However, in most DSS queries, all fragments are read.
- If you are using expressions, create them so that I/O requests, rather than quantities of data, are balanced across disks. For example, if the majority of your queries access only a portion of data in the table, set up your fragmentation expression to spread active portions of the table across disks, even if this arrangement results in an uneven distribution of rows.
- Keep fragmentation expressions simple. Fragmentation expressions can be as complex as you want. However, complex expressions take more time to evaluate and might prevent fragments from being eliminated from queries.

- Arrange fragmentation expressions so that the most restrictive condition for each dbspace is tested within the expression first. When the database server tests a value against the criteria for a given fragment, evaluation stops when a condition for that fragment tests false. Thus, if the condition that is most likely to be false is placed first, fewer conditions need to be evaluated before the database server moves to the next fragment. For example, in the following expression, the database server tests all six of the inequality conditions when it attempts to insert a row with a value of 25:

  ```
  x >= 1 and x <= 10 in dbspace1,
  x > 10 and x <= 20 in dbspace2,
  x > 20 and x <= 30 in dbspace3
  ```

  By comparison, only four conditions in the following expression need to be tested: the first inequality for **dbspace1** (`x <= 10`), the first for **dbspace2** (`x <= 20`), and both conditions for **dbspace3**:

  ```
  x <= 10 and x >= 1 in dbspace1,
  x <= 20 and x > 10 in dbspace2,
  x <= 30 and x > 20 in dbspace3
  ```

- Avoid any expression that requires a data-type conversion. Type conversions increase the time to evaluate the expression. For instance, a DATE data type is implicitly converted to INTEGER for comparison purposes.
- Do not fragment on columns that change frequently unless you are willing to incur the administration costs. For example, if you fragment on a date column and older rows are deleted, the fragment with the oldest dates tends to empty, and the fragment with the recent dates tends to fill up. Eventually you have to drop the old fragment and add a new fragment for newer orders.
- Do not fragment every table. Identify the critical tables that are accessed most frequently. Put only one fragment for a table on a disk.
- Do not fragment small tables. Fragmenting a small table across many disks might not be worth the overhead of starting all the scan threads to access the fragments. Also, balance the number of fragments with the number of processors on your system.
- When you define a fragmentation strategy on an unfragmented table, check the next-extent size to ensure that you are not allocating large amounts of disk space for each fragment.

Fragmenting Indexes

When you fragment a table, the indexes that are associated with that table are fragmented implicitly, according to the fragmentation scheme that you use. You can also use the FRAGMENT BY EXPRESSION clause of the CREATE INDEX statement to fragment the index for any table explicitly. Each index of a fragmented table occupies its own tblspace with its own extents.

You can fragment the index with either of the following strategies:

- Same fragmentation strategy as the table
- Different fragmentation strategy from the table

Attached Indexes

An *attached index* is an index that implicitly follows the table fragmentation strategy (distribution scheme and set of dbspaces in which the fragments are located). The database server automatically creates an attached index when you first fragment the table.

To create an attached index, do not specify a fragmentation strategy or storage option in the CREATE INDEX statement, as in the following sample SQL statements:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN dbspace1,
      (a >=5 AND a < 10) IN dbspace2
      ...
   ;

CREATE INDEX idx1 ON tb1(a);
```

The database server fragments the attached index according to the same distribution scheme as the table by using the same rule for index keys as for table data. As a result, attached indexes have the following physical characteristics:

- The number of index fragments is the same as the number of data fragments.
- Each attached index fragment resides in the same dbspace as the corresponding table data, but in a separate tblspace.
- An attached index or an index on a nonfragmented table uses 4 bytes for the row pointer for each index entry. For more information on how to estimate space for an index, refer to "Estimating lndex Pages" on page 7-3.

Detached Indexes

A *detached index* is an index with a separate fragmentation strategy that you set up explicitly with the CREATE INDEX statement, as in the following sample SQL statements:

```
CREATE TABLE tb1 (a int)
      FRAGMENT BY EXPRESSION
         (a <= 10) IN tabdbspc1,
         (a <= 20) IN tabdbspc2,
         (a <= 30) IN tabdbspc3;

CREATE INDEX idx1 ON tb1 (a)
      FRAGMENT BY EXPRESSION
         (a <= 10) IN idxdbspc1,
         (a <= 20) IN idxdbspc2,
         (a <= 30) IN idxdbspc3;
```

This example illustrates a common fragmentation strategy, to fragment indexes in the same way as the tables, but specify different dbspaces for the index fragments. This fragmentation strategy of putting the index fragments in different dbspaces from the table can improve the performance of operations such as backup, recovery, and so forth.

If you do not want to fragment the index, you can put the entire index in a separate dbspace.

You can fragment the index for any table by expression. However, you cannot explicitly create a round-robin fragmentation scheme for an index. Whenever you fragment a table using a round-robin fragmentation scheme, Informix recommends that you convert all indexes that accompany the table to detached indexes for the best performance.

Detached indexes have the following physical characteristics:

- Each detached index fragment resides in a different tblspace from the corresponding table data. Therefore, the data and index pages cannot be interleaved within the tblspace.
- Attached index fragments have their own extents and *tblspace IDs*. The tblspace ID is also known as the *fragment ID* and *partition number*. A detached index uses 8 bytes of disk space per index entry for the fragment ID and row pointer combination. For more information on how to estimate space for an index, refer to "Estimating lndex Pages" on page 7-3.

The database server stores the location of each table and index fragment, along with other related information, in the system catalog table **sysfragments**. You can use the **sysfragments** system catalog table to access the following information about fragmented tables and indexes:

- The value in the **partn** field is the partition number or fragment ID of the table or index fragment. The partition number for a detached index is different from the partition number of the corresponding table fragment.
- The value in the **strategy** field is the distribution scheme used in the fragmentation strategy.

For a complete description of field values that this **sysfragments** system catalog table contains, refer to the *Informix Guide to SQL: Reference*. For information on how to use **sysfragments** to monitor your fragments, refer to "Monitoring Fragment Use" on page 9-39.

Restrictions on Indexes for Fragmented Tables

If the database server scans a fragmented index, multiple index fragments must be scanned and the results merged together. (The exception is if the index is fragmented according to some index-key range rule, and the scan does not cross a fragment boundary.) Because of this requirement, performance on index scans might suffer if the index is fragmented.

Because of these performance considerations, the database server places the following restrictions on indexes:

- You cannot fragment indexes by round-robin.
- You cannot fragment unique indexes by an expression that contains columns that are not in the index key.

For example, the following statement is not valid:

```
CREATE UNIQUE INDEX ia on tab1(col1)
   FRAGMENT BY EXPRESSION
      col2<10 in dbsp1,
      col2>=10 AND col2<100 in dbsp2,
      col2>100 in dbsp3;
```

Fragmenting Temporary Tables

You can fragment an explicit temporary table across dbspaces that reside on different disks. For more information on explicit and implicit temporary tables, refer to your *Administrator's Guide*.

You can create a temporary, fragmented table with the TEMP TABLE clause of the CREATE TABLE statement. However, you cannot alter the fragmentation strategy of fragmented temporary tables (as you can with permanent tables). The database server deletes the fragments that are created for a temporary table at the same time that it deletes the temporary table.

You can define your own fragmentation strategy for an explicit temporary table, or you can let the database server dynamically determine the fragmentation strategy.

Using Distribution Schemes to Eliminate Fragments

Fragment elimination is a database server feature that reduces the number of fragments involved in a database operation. This capability can improve performance significantly and reduce contention for the disks on which fragments reside.

Fragment elimination improves both response time for a given query and concurrency between queries. Because the database server does not need to read in unnecessary fragments, I/O for a query is reduced. Activity in the LRU queues is also reduced.

If you use an appropriate distribution scheme, the database server can eliminate fragments from the following database operations:

- The fetch portion of the SELECT, INSERT, DELETE or UPDATE statements in SQL

  The database server can eliminate fragments when these SQL statements are optimized, before the actual search.

- Nested-loop joins

  When the database server obtains the key value from the outer table, it can eliminate fragments to search on the inner table.

Whether the database server can eliminate fragments from a search depends on two factors:

- The distribution scheme in the fragmentation strategy of the table that is being searched
- The form of the query expression (the expression in the WHERE clause of a SELECT, INSERT, DELETE or UPDATE statement)

Fragmentation Expressions for Fragment Elimination

When the fragmentation strategy is defined with any of the following operators, fragment elimination can occur for a query on the table.

```
IN
=
<
>
<=
>=
AND
OR
NOT
MATCH
LIKE
```

If the fragmentation expression uses any of the following operators, fragment elimination does not occur for queries on the table.

```
!=
IS NULL
IS NOT NULL
```

For examples of fragmentation expressions that allow fragment elimination, refer to "Effectiveness of Fragment Elimination" on page 9-24.

Query Expressions for Fragment Elimination

A query expression (the expression in the WHERE clause) can consist of any of the following expressions:

- Simple expression
- Not simple expression
- Multiple expressions

The database server considers only simple expressions or multiple simple expressions combined with certain operators for fragment elimination.

A simple expression consists of the following parts:

```
column operator value
```

Simple Expression Part	Description
column	Is a single column name The database server supports fragment elimination on all column types except columns that are defined with the NCHAR, NVARCHAR, BYTE, and TEXT data types.
operator	Must be an equality or range operator
value	Must be a literal or a host variable

The following examples show simple expressions:

```
name = "Fred"
date < "01/25/1994"
value >= :my_val
```

The following examples are not simple expressions:

```
unitcost * count > 4500
price <= avg(price)
result + 3 > :limit
```

The database server considers two types of simple expressions for fragment elimination, based on the operator:

- Range expressions
- Equality expressions

Range Expressions in Query

Range expressions use the following relational operators:

```
<, >, <=, >=, !=
```

The database server can handle one or two column fragment elimination on queries with any combination of these relational operators in the WHERE clause.

The database server can also eliminate fragments when these range expressions are combined with the following operators:

```
AND, OR, NOT
IS NULL, IS NOT NULL
MATCH, LIKE
```

If the range expression contains MATCH or LIKE, the database server can also eliminate fragments if the string ends with a wildcard character. The following examples show query expressions that can take advantage of fragment elimination:

```
columna MATCH "ab*"
columna LIKE "ab%" OR columnb LIKE "ab*"
```

Equality Expressions in Query

Equality expressions use the following equality operators:

```
=, IN
```

The database server can handle one or multiple column fragment elimination on queries with a combination of these equality operators in the WHERE clause. The database server can also eliminate fragments when these equality expressions are combined with the following operators:

```
AND, OR
```

Effectiveness of Fragment Elimination

The database server cannot eliminate fragments when you fragment a table with a round-robin distribution scheme. Furthermore, not all expression-based distribution schemes give you the same fragment-elimination behavior.

Figure 9-2 summarizes the fragment-elimination behavior for different combinations of expression-based distribution schemes and query expressions.

Figure 9-2
Fragment Elimination for Different Categories of Expression-Based Distribution Schemes and Query Expressions

	Type of Expression-Based Distribution Scheme		
Type of Query (WHERE clause) Expression	**Nonoverlapping Fragments on a Single Column**	**Overlapping or Non-contiguous Fragments on a Single Column**	**Nonoverlapping Fragments on Multiple Columns**
Range expression	Fragments can be eliminated.	Fragments cannot be eliminated.	Fragments cannot be eliminated.
Equality expression	Fragments can be eliminated.	Fragments can be eliminated.	Fragments can be eliminated.

Figure 9-2 indicates that the distribution schemes enable fragment elimination, but the effectiveness of fragment elimination is determined by the WHERE clause of the query in question.

For example, consider a table fragmented with the following expression:

```
FRAGMENT BY EXPRESSION
100 < column_a AND column_b < 0 IN dbsp1,
100 >= column_a AND column_b < 0 IN dbsp2,
column_b >= 0 IN dbsp3
```

The database server cannot eliminate any fragments from the search if the WHERE clause has the following expression:

```
column_a = 5 OR column_b = -50
```

On the other hand, the database server can eliminate the fragment in dbspace **dbsp3** if the WHERE clause has the following expression:

```
column_b = -50
```

Furthermore, the database server can eliminate the two fragments in dbspaces **dbsp2** and **dbsp3** if the WHERE clause has the following expression:

```
column_a = 5 AND column_b = -50
```

The following sections discuss distribution schemes to fragment data to improve fragment elimination behavior.

Nonoverlapping Fragments on a Single Column

A fragmentation rule that creates nonoverlapping fragments on a single column is the preferred fragmentation rule from a fragment-elimination standpoint. The advantage of this type of distribution scheme is that the database server can eliminate fragments for queries with range expressions as well as queries with equality expressions. Informix recommends that you meet these conditions when you design your fragmentation rule. Figure 9-3 gives an example of this type of fragmentation rule.

```
...
FRAGMENT BY EXPRESSION
a <= 8 OR a IN (9,10) IN dbsp1,
10 < a AND a <= 20 IN dbsp2,
a IN (21,22, 23) IN dbsp3,
a > 23 IN dbsp4;
```

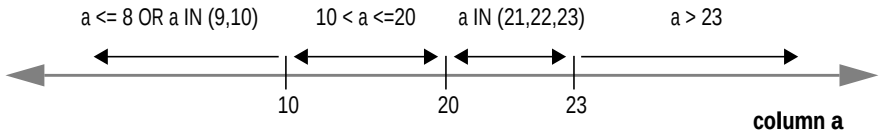


Figure 9-3
Schematic Example of Nonoverlapping Fragments on a Single Column

You can create nonoverlapping fragments using a range rule or an arbitrary rule based on a single column. You can use relational operators, as well as AND, IN, OR, or BETWEEN. Be careful when you use the BETWEEN operator. When the database server parses the BETWEEN keyword, it includes the end points that you specify in the range of values. Avoid using a REMAINDER clause in your expression. If you use a REMAINDER clause, the database server cannot always eliminate the remainder fragment.

Overlapping Fragments on a Single Column

The only restriction for this category of fragmentation rule is that you base the fragmentation rule on a single column. The fragments can be overlapping and noncontiguous. You can use any range, MOD function, or arbitrary rule that is based on a single column. Figure 9-4 shows an example of this type of fragmentation rule.

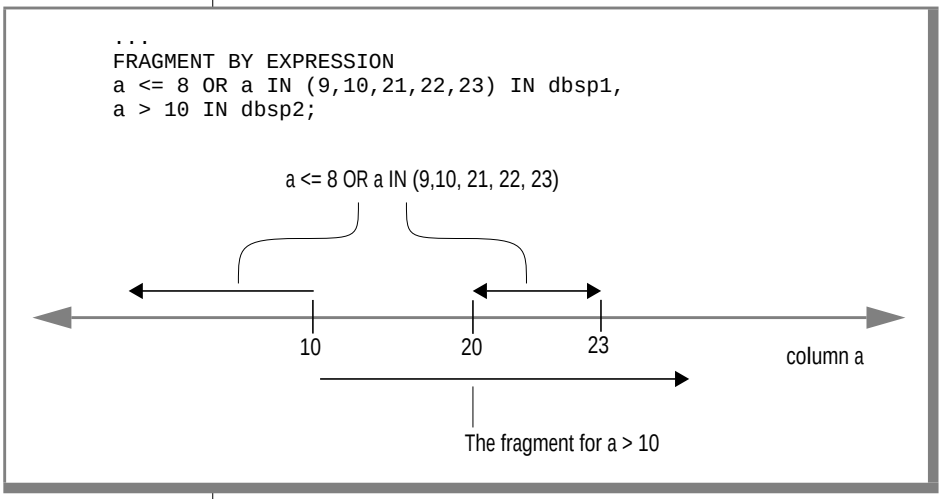


Figure 9-4
Schematic Example of Overlapping Fragments on a Single Column

If you use this type of distribution scheme, the database server can eliminate fragments on an equality search but not a range search. This distribution scheme can still be useful because all INSERT and many UPDATE operations perform equality searches.

This alternative is acceptable if you cannot use an expression that creates nonoverlapping fragments with contiguous values. For example, in cases where a table is growing over time, you might want to use a MOD function rule to keep the fragments of similar size. Expression-based distribution schemes that use MOD function rules fall into this category because the values in each fragment are not contiguous.

Nonoverlapping Fragments, Multiple Columns

This category of expression-based distribution scheme uses an arbitrary rule to define nonoverlapping fragments based on multiple columns. Figure 9-5 shows an example of this type of fragmentation rule.

```
...
FRAGMENT BY EXPRESSION
0 < a AND a <= 10 AND b IN ('E', 'F','G') IN dbsp1,
0 < a AND a <= 10 AND b IN ('H', 'I','J') IN dbsp2,
10 < a AND a <= 20 AND b IN ('E', 'F','G') IN dbsp3,
10 < a AND a <= 20 AND b IN ('H', 'I','J') IN dbsp4,
20 < a AND a <= 30 AND b IN ('E', 'F','G') IN dbsp5,
20 < a AND a <= 30 AND b IN ('H', 'I','J') IN dbsp6;
```

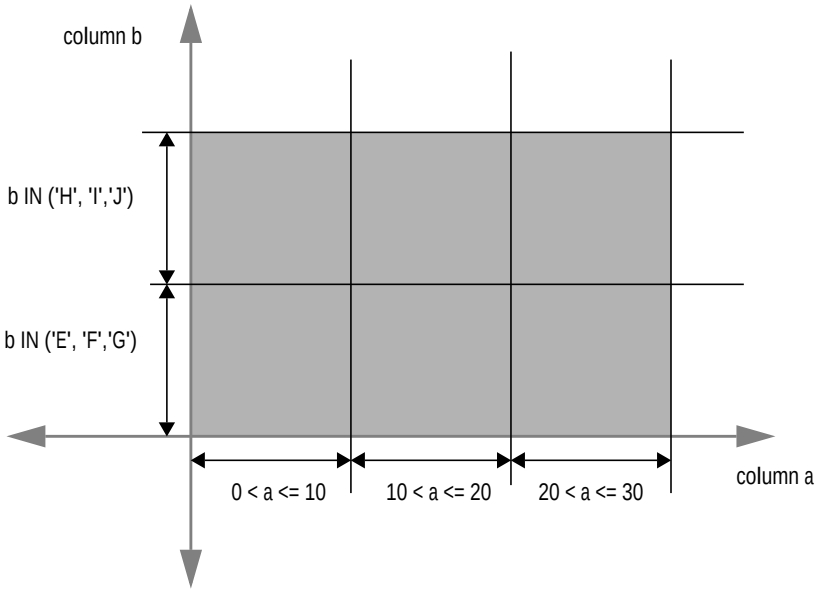


Figure 9-5
Schematic Example of Nonoverlapping Fragments on Two Columns

If you use this type of distribution scheme, the database server can eliminate fragments on an equality search but not a range search. This capability can still be useful because all INSERT operations and many UPDATE operations perform equality searches. Avoid using a REMAINDER clause in the expression. If you use a REMAINDER clause, the database server cannot always eliminate the remainder fragment.

This alternative is acceptable if you cannot obtain sufficient granularity using an expression based on a single column.

Improving the Performance of Attaching and Detaching Fragments

Many users use ALTER FRAGMENT ATTACH and DETACH statements to add or remove a large amount of data in a very large table. ALTER FRAGMENT DETACH provides a way to delete a segment of the table data rapidly. Similarly, ALTER FRAGMENT ATTACH provides a way to load large amounts of data incrementally into an existing table by taking advantage of the fragmentation technology. However, the ALTER FRAGMENT ATTACH and DETACH statements can take a long time to execute when the database server rebuilds indexes on the surviving table.

The database server provides performance optimizations for the ALTER FRAGMENT ATTACH and DETACH statements that cause the database server to reuse the indexes on the surviving tables. Therefore, the database server can eliminate the index build during the attach or detach operation, which:

- Reduces the time that it takes for the ALTER FRAGMENT ATTACH and ALTER FRAGMENT DETACH statements to execute
- Improves the table availability

Improving ALTER FRAGMENT ATTACH Performance

To take advantage of these performance optimizations for the ALTER FRAGMENT ATTACH statement, you must meet all of the following requirements:

- Formulate appropriate distribution schemes for your table and index fragments.
- Ensure that no data movement occurs between the resultant partitions due to fragment expressions.
- Update statistics for all the participating tables.
- Make the indexes on the attached tables unique if the index on the surviving table is unique.

Important: *Only logging databases can benefit from the performance improvements for the ALTER FRAGMENT ATTACH statement. Without logging, the database server works with multiple copies of the same table to ensure recoverability of the data when a failure occurs. This requirement prevents reuse of the existing index fragments.*

Formulating Appropriate Distribution Schemes

This section describes three distribution schemes that allow the attach operation of the ALTER FRAGMENT statement to reuse existing indexes:

- Fragment the index in the same way as the table.
- Fragment the index with the same set of fragment expressions as the table.
- Attach unfragmented tables to form a fragmented table.

Fragmenting the Index in the Same Way as the Table

You fragment an index in the same way as the table when you create an index without specifying a fragmentation strategy. A fragmentation strategy is the distribution scheme and set of dbspaces in which the fragments are located. For details, refer to "Planning a Fragmentation Strategy" on page 9-3.

For example, suppose you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2;

CREATE INDEX idx1 ON tb1(a);
```

Suppose you then create another table that is not fragmented, and you subsequently decide to attach it to the fragmented table.

```
CREATE TABLE tb2 (a int, CHECK (a >=10 AND a<15))
   IN db3;

CREATE INDEX idx2 ON tb2(a)
   IN db3;

ALTER FRAGMENT ON TABLE tb1
      ATTACH
         tb2 AS (a >= 10 and a<15) AFTER db2;
```

This attach operation can take advantage of the existing index **idx2** if no data movement occurs between the existing and the new table fragments. If no data movement occurs:

- The database server reuses index **idx2** and converts it to a fragment of index **idx1**.
- The index **idx1** remains as an index with the same fragmentation strategy as the table **tb1**.

If the database server discovers that one or more rows in the table **tb2** belong to preexisting fragments of the table **tb1**, the database server:

- Drops and rebuilds the index **idx1** to include the rows that were originally in tables **tb1** and **tb2**
- Drops the index **idx2**

For more information on how to ensure no data movement between the existing and the new table fragments, refer to "Ensuring No Data Movement When You Attach a Fragment" on page 9-33.

Fragmenting the Index with the Same Distribution Scheme as the Table

You fragment an index with the same distribution scheme as the table when you create the index that uses the same fragment expressions as the table.

The database server determines if the fragment expressions are identical, based on the equivalency of the expression tree instead of the algebraic equivalence. For example, consider the following two expressions:

```
(col1 >= 5)
(col1 = 5 OR col1 > 5)
```

Although these two expressions are algebraically equivalent, they are not identical expressions.

Suppose you create two fragmented tables and indexes with the following SQL statements:

```
CREATE TABLE tb1 (a INT)
   FRAGMENT BY EXPRESSION
      (a <= 10) IN tabdbspc1,
      (a <= 20) IN tabdbspc2,
      (a <= 30) IN tabdbspc3;
CREATE INDEX idx1 ON tb1 (a)
   FRAGMENT BY EXPRESSION
      (a <= 10) IN idxdbspc1,
      (a <= 20) IN idxdbspc2,
      (a <= 30) IN idxdbspc3;

CREATE TABLE tb2 (a INT CHECK a> 30 AND a<= 40)
   IN tabdbspc4;
CREATE INDEX idx2 ON tb2(a)
   IN idxdbspc4;
```

Suppose you then attach table **tb2** to table **tb1** with the following sample SQL statement:

```
ALTER FRAGMENT ON TABLE tb1
   ATTACH tb2 AS (a <= 40);
```

The database server can eliminate the rebuild of index **idx1** for this attach operation for the following reasons:

- The fragmentation expression for index **idx1** is identical to the fragmentation expression for table **tb1**. The database server:
  - Expands the fragmentation of the index **idx1** to the dbspace **idxdbspc4**
  - Converts index **idx2** to a fragment of index **idx1**
- No rows move from one fragment to another because the CHECK constraint is identical to the resulting fragmentation expression of the attached table.

  For more information on how to ensure no data movement between the existing and the new table fragments, refer to "Ensuring No Data Movement When You Attach a Fragment" on page 9-33.

Attaching Unfragmented Tables Together

You also take advantage of the performance improvements for the ALTER FRAGMENT ATTACH operation when you combine two unfragmented tables into one fragmented table.

For example, suppose you create two unfragmented tables and indexes with the following SQL statements:

```
CREATE TABLE tb1(a int) IN db1;
   CREATE INDEX idx1 ON tb1(a) in db1;
CREATE TABLE tb2(a int) IN db2;
   CREATE INDEX idx2 ON tb2(a) in db2;
```

You might want to combine these two unfragmented tables with the following sample distribution scheme:

```
ALTER FRAGMENT ON TABLE tb1
   ATTACH
      tb1 AS (a <= 100),
      tb2 AS (a > 100);
```

If no data migrates between the fragments of **tb1** and **tb2**, the database server redefines index **idx1** with the following fragmentation strategy:

```
CREATE INDEX idx1 ON tb1(a) F
   FRAGMENT BY EXPRESSION
      a <= 100 IN db1,
      a > 100 IN db2;
```

Important: *This behavior results in a different fragmentation strategy for the index than prior to Version 7.3 and Version 9.2 of the database server. In earlier versions, the ALTER FRAGMENT ATTACH statement creates an unfragmented detached index in the dbspace* ***db1****.*

Ensuring No Data Movement When You Attach a Fragment

To ensure that no data movement occurs, take the following steps:

To ensure no data movement

1. Establish a check constraint on the attached table that is identical to the fragment expression that it will assume after the ALTER FRAGMENT ATTACH operation.
2. Define the fragments with nonoverlapping expressions.

For example, you might create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2;

CREATE INDEX idx1 ON tb1(a);
```

Suppose you create another table that is not fragmented, and you subsequently decide to attach it to the fragmented table.

```
CREATE TABLE tb2 (a int, check (a >=10 and a<15))
   IN db3;

CREATE INDEX idx2 ON tb2(a)
   IN db3;

ALTER FRAGMENT ON TABLE tb1
   ATTACH
      tb2 AS (a >= 10 AND a<15) AFTER db2;
```

This ALTER FRAGMENT ATTACH operation takes advantage of the existing index **idx2** because the following steps were performed in the example to prevent data movement between the existing and the new table fragment:

- The check constraint expression in the CREATE TABLE **tb2** statement is identical to the fragment expression for table **tb2** in the ALTER FRAGMENT ATTACH statement.
- The fragment expressions specified in the CREATE TABLE **tb1** and the ALTER FRAGMENT ATTACH statements are not overlapping.

Therefore, the database server preserves index **idx2** in dbspace **db3** and converts it into a fragment of index **idx1**. The index **idx1** remains as an index with the same fragmentation strategy as the table **tb1**.

Updating Statistics on All Participating Tables

The database server tries to reuse the indexes on the attached tables as fragments of the resultant index. However, the corresponding index on the attached table might not exist or might not be usable due to disk-format mismatches. In these cases, it might be faster to build an index on the attached tables rather than to build the entire index on the resultant table.

The database server estimates the cost to create the whole index on the resultant table. The database server then compares this cost to the cost of building the individual index fragments for the attached tables and chooses the index build with the least cost.

To ensure the correctness of the cost estimates, Informix recommends that you execute the UPDATE STATISTICS statement on all of the participating tables before you attach the tables. The LOW mode of the UPDATE STATISTICS statement is sufficient to derive the appropriate statistics for the optimizer to determine cost estimates for rebuilding indexes.

Corresponding Index Does Not Exist

Suppose you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int, b int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2;
CREATE INDEX idx1 ON tb1(a);
```

Suppose you then create two more tables that are not fragmented, and you subsequently decide to attach them to the fragmented table.

```
CREATE TABLE tb2 (a int, b int, check (a >=10 and a<15))
   IN db3;
CREATE INDEX idx2 ON tb2(a)
   IN db3;

CREATE TABLE tb3 (a int, b int, check (a >= 15 and a<20))
   IN db4;
CREATE INDEX idx3 ON tb3(b)
   IN db4;

UPDATE STATISTICS FOR TABLE tb1;
UPDATE STATISTICS FOR TABLE tb2;
UPDATE STATISTICS FOR TABLE tb3;

ALTER FRAGMENT ON TABLE tb1
   ATTACH
      tb2 AS (a >= 10 and a<15)
      tb3 AS (a >= 15 and a<20);
```

In the preceding example, table **tb3** does not have an index on column **a** that can serve as the fragment of the resultant index **idx1**. The database server estimates the cost of building the index fragment for column **a** on the consumed table **tb3** and compares this cost to rebuilding the entire index for all fragments on the resultant table. The database server chooses the index build with the least cost.

Index on Table Is Not Usable

Suppose you create tables and indexes as in the previous section, but the index on the third table specifies a dbspace that the first table also uses. The following SQL statements show this scenario:

```
CREATE TABLE tb1(a int, b int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2;
CREATE INDEX idx1 ON tb1(a);
CREATE TABLE tb2 (a int, b int, check (a >=10 and a<15))
   IN db3;
CREATE INDEX idx2 ON tb2(a)
   IN db3;

CREATE TABLE tb3 (a int, b int, check (a >= 15 and a<20))
   IN db4;
CREATE INDEX idx3 ON tb3(a)
   IN db2 ;
```

This example creates the index **idx3** on table **tb3** in the dbspace **db2**. As a result, index **idx3** is not usable because index **idx1** already has a fragment in the dbspace **db2,** and the fragmentation strategy does not allow more than one fragment to be specified in a given dbspace.

Again, the database server estimates the cost of building the index fragment for column **a** on the consumed table **tb3** and compares this cost to rebuilding the entire index **idx1** for all fragments on the resultant table. Then the database server chooses the index build with the least cost.

Improving ALTER FRAGMENT DETACH Performance

To take advantage of the performance improvements for the ALTER FRAGMENT DETACH statement, formulate appropriate distribution schemes for your table and index fragments.

To eliminate the index build during execution of the ALTER FRAGMENT DETACH statement, use one of the following fragmentation strategies:

- Fragment the index in the same way as the table.
- Fragment the index with the same distribution scheme as the table.

Important: *Only logging databases can benefit from the performance improvements for the ALTER FRAGMENT DETACH statement. Without logging, the database server works with multiple copies of the same table to ensure recoverability of the data when a failure occurs. This requirement prevents reuse of the existing index fragments.*

Fragmenting the Index in the Same Way as the Table

You fragment an index in the same way as the table when you create a fragmented table and subsequently create an index without specifying a fragmentation strategy.

For example, suppose you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2,
      (a >=10 AND a <15) IN db3;
CREATE INDEX idx1 ON tb1(a);
```

The database server fragments the index keys into dbspaces **db1**, **db2**, and **db3** with the same column **a** value ranges as the table because the CREATE INDEX statement does not specify a fragmentation strategy.

Suppose you then decide to detach the data in the third fragment with the following SQL statement:

```
ALTER FRAGMENT ON TABLE tb1
   DETACH db3 tb3;
```

Because the fragmentation strategy of the index is the same as the table, the ALTER FRAGMENT DETACH statement does not rebuild the index after the detach operation. The database server drops the fragment of the index in dbspace **db3**, updates the system catalog tables, and eliminates the index build.

Fragmenting the Index Using Same Distribution Scheme as the Table

You fragment an index with the same distribution scheme as the table when you create the index that uses the same fragment expressions as the table.

A common fragmentation strategy is to fragment indexes in the same way as the tables but to specify different dbspaces for the index fragments. This fragmentation strategy of putting the index fragments into different dbspaces from the table can improve the performance of operations such as backup, recovery, and so forth.

For example, suppose you create a fragmented table and index with the following SQL statements:

```
CREATE TABLE tb1(a int, b int)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a < 5) IN db1,
      (a >=5 AND a <10) IN db2,
      (a >=10 AND a <15) IN db3;

CREATE INDEX idx1 on tb1(a)
   FRAGMENT BY EXPRESSION
      (a >=0 AND a< 5) IN db4,
      (a >=5 AND a< 10) IN db5,
      (a >=10 AND a<15) IN db6;
```

Suppose you then decide to detach the data in the third fragment with the following SQL statement:

```
ALTER FRAGMENT ON TABLE tb1
   DETACH db3 tb3;
```

Because the distribution scheme of the index is the same as the table, the ALTER FRAGMENT DETACH statement does not rebuild the index after the detach operation. The database server drops the fragment of the index in dbspace **db3**, updates the system catalog tables, and eliminates the index build.

Monitoring Fragment Use

Once you determine a fragmentation strategy, you can monitor fragmentation in the following ways:

- Run individual **onstat** utility commands to capture information about specific aspects of a running query.
- Execute a SET EXPLAIN statement before you run a query to write the query plan to an output file.

Using the onstat Utility

You can monitor I/O activity to verify your strategy and determine whether I/O is balanced across fragments.

The **onstat -g ppf** command displays the number of read-and-write requests sent to each fragment that is currently open. Because a request can trigger multiple I/O operations, these requests do not indicate how many individual disk I/O operations occur, but you can get a good idea of the I/O activity from these columns.

However, the output by itself does not show in which table a fragment is located. To determine the table for the fragment, join the `partnum` column in the output to the **partnum** column in the **sysfragments** system catalog table. The **sysfragments** table displays the associated **table id**. To determine the table name for the fragment, join the **table id** column in **sysfragments** to the **table id** column in **systables**.

To determine the table name

1. Obtain the value in the **partnum** field of the **onstat -g ppf** output.
2. Join the **tabid** column in the **sysfragments** system catalog table with the **tabid** column in the **systables** system catalog table to obtain the table name from **systables**.

   Use the **partnum** field value that you obtain in step 1 in the SELECT statement.

```
SELECT a.tabname FROM systables a, sysfragments b
   WHERE a.tabid = b.tabid
      AND partn = partnum_value;
```

Using SET EXPLAIN

When the table is fragmented, the output of the SET EXPLAIN ON statement shows which table or index the database server scans to execute the query. The SET EXPLAIN output identifies the fragments with a fragment number. The fragment numbers are the same as those contained in the **partn** column in the **sysfragments** system catalog table.

The following example of SET EXPLAIN output shows a query that takes advantage of fragment elimination and scans two fragments in table **t1**:

```
QUERY:
------
SELECT * FROM t1 WHERE c1 > 12

Estimated Cost: 3
Estimated # of Rows Returned: 2

1) informix.t1: SEQUENTIAL SCAN (Serial, fragments: 1, 2)

   Filters: informix.t1.c1 > 12
```

If the optimizer must scan all fragments (that is, if it is unable to eliminate any fragment from consideration), the SET EXPLAIN output displays `fragments: ALL`. In addition, if the optimizer eliminates all the fragments from consideration (that is, none of the fragments contain the queried information), the SET EXPLAIN output displays `fragments: NONE`. For information on how the database server eliminates a fragment from consideration, refer to "Using Distribution Schemes to Eliminate Fragments" on page 9-21.

For more information on the SET EXPLAIN ON statement, refer to "Query Plan Report" on page 10-12.

Chapter

10

Queries and the Query Optimizer

In This Chapter 10-3

The Query Plan 10-3
- Access Plan 10-4
- Join Plan 10-4
  - Nested-Loop Join 10-5
  - Hash Join 10-6
  - Join Order 10-7
- Example of Query-Plan Execution 10-8
  - Join with Column Filters 10-9
  - Join with Indexes 10-10
- Query Plan Evaluation 10-12
- Query Plan Report. 10-12
  - EXPLAIN Output File 10-13
  - EXPLAIN Output Description 10-13
- Sample Query Plan Reports 10-15
  - Single-Table Query 10-15
  - Multitable Query 10-16
  - Key-First Scan 10-17
  - Query Plans for Subqueries 10-18
  - Query Plans for Collection-Derived Tables 10-19

Factors That Affect the Query Plan 10-21
- Statistics Held for the Table and Index 10-21
- Filters in the Query 10-23
- Indexes for Evaluating a Filter 10-24
- Effect of PDQ on the Query Plan 10-25
- Effect of OPTCOMPIND on the Query Plan 10-26
  - Single-Table Query 10-26
  - Multitable Query 10-26
- Effect of Available Memory on the Query Plan 10-27

Time Costs of a Query 10-27
Memory-Activity Costs 10-28
Sort-Time Costs 10-28
Row-Reading Costs 10-30
Sequential Access Costs 10-31
Nonsequential Access Costs 10-31
Index Lookup Costs 10-32
Reading Duplicate Values From an Index 10-32
Searching for NCHAR or NVARCHAR Columns in an Index 10-32
In-Place ALTER TABLE Costs 10-33
View Costs 10-33
Small-Table Costs 10-34
Data-Mismatch Costs 10-35
GLS Functionality Costs 10-36
Network-Access Costs 10-36

SQL Within SPL Routines. 10-38
SQL Optimization 10-38
Displaying the Execution Plan 10-39
Automatic Reoptimization. 10-40
Reoptimizing SPL Routines 10-40
Optimization Levels for SQL in SPL Routines 10-41
Execution of an SPL Routine 10-41
UDR Cache 10-42
Changing the UDR Cache 10-42
Monitoring the UDR Cache 10-43

Trigger Execution 10-44
Performance Implications for Triggers 10-45
SELECT Triggers on Tables in a Table Hierarchy 10-46
SELECT Triggers and Row Buffering 10-46

In This Chapter

This chapter explains how the database server manages query optimization and factors that you can influence to affect the query plan. Performance considerations for SPL routine, the UDR cache, and triggers is also covered.

The parallel database query (PDQ) features in the database server provide the largest potential performance improvements for a query. Chapter 12, "Parallel Database Query," describes PDQ and the Memory Grant Manager (MGM) and explains how to control resource use by queries.

PDQ provides the most substantial performance gains if you fragment your tables as described in Chapter 9, "Fragmentation Guidelines."

Chapter 13, "Improving Individual Query Performance," explains how to improve the performance of specific queries.

The Query Plan

The query optimizer formulates a *query plan* to fetch the data rows that are required to process a query.

The optimizer must evaluate the different ways in which a query might be performed. For example, the optimizer must determine whether indexes should be used. If the query includes a join, the optimizer must determine the join plan (hash or nested loop) and the order in which tables are evaluated or joined. The following section explains the components of a query plan.

Access Plan

The way that the optimizer chooses to read a table is called an *access plan*. The simplest method to access a table is to read it sequentially, which is called a *table scan*. The optimizer chooses a table scan when most of the table must be read or the table does not have an index that is useful for the query.

The optimizer can also choose to access the table by an index. If the column in the index is the same as a column in a filter of the query, the optimizer can use the index to retrieve only the rows that the query requires. The optimizer can use a *key-only index scan* if the columns requested are within one index on the table. The database server retrieves the needed data from the index and does not access the associated table.

Important: *The optimizer does not choose a key-only scan for a VARCHAR column. If you want to take advantage of key-only scans, use the ALTER TABLE with the MODFIY clause to change the column to a CHAR data type.*

The optimizer compares the cost of each plan to determine the best one. The database server derives cost from estimates of the number of I/O operations required, calculations to produce the results, rows accessed, sorting, and so forth.

Join Plan

When a query contains more than one table, the database server joins them using filters in the query. For example, in the following query, the customer and orders table are joined by the `customer.customer_num = orders.customer_num` filter:

```
SELECT * from customer, orders
WHERE customer.customer_num = orders.customer_num
AND customer.lname = "Higgins";
```

The way that the optimizer chooses to join the tables is the *join plan*. The join method can be a nested-loop join or a hash join.

Because of the nature of hash joins, an application with isolation level set to Repeatable Read might temporarily lock all the records in tables that are involved in the join, including records that fail to qualify the join. This situation leads to decreased concurrency among connections. Conversely, nested-loop joins lock fewer records but provide reduced performance when a large number of rows are accessed. Thus, each join method has advantages and disadvantages.

Nested-Loop Join

In a nested-loop join, the database server scans the first, or *outer table*, and then joins each of the rows that pass table filters to the rows found in the second, or *inner table.* (Refer to Figure 10-1 on page 10-5.) The database server accesses an outer table by an index or by a table scan. The database server applies any table filters first. For each row that satisfies the filters on the outer table, the database server reads the inner table to find a match.

The database server reads the inner table once for every row in the outer table that fulfills the table filters. Because of the potentially large number of times that the inner table can be read, the database server usually accesses the inner table by an index.

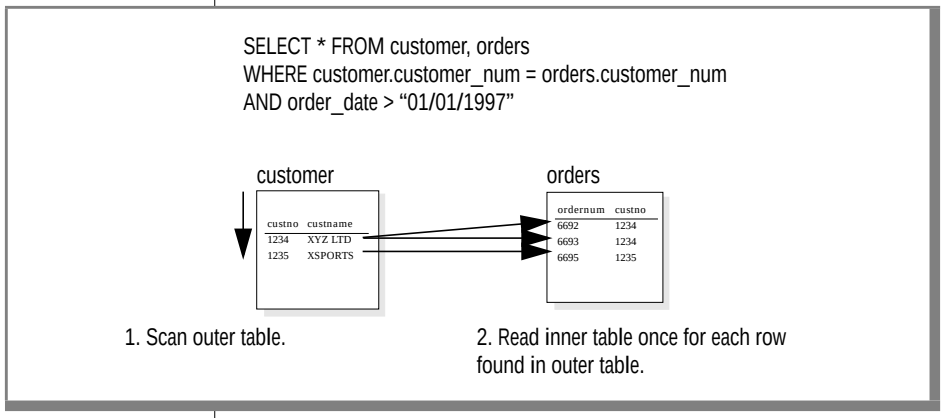


Figure 10-1
Nested-Loop Join

If the inner table does not have an index, the database server might construct an *autoindex* at the time of query execution. The optimizer might determine that the cost to construct an *autoindex* at the time of query execution is less than the cost to scan the inner table for each qualifying row in the outer table.

If the optimizer changes a subquery to a nested-loop join, it might use a variation of the nested-loop join, called a *semi join*. In a semi join, the database server reads the inner table only until it finds a match. In other words, for each row in the outer table, the inner table contributes at most one row. For more information on how the optimizer handles subqueries, refer to "Query Plans for Subqueries" on page 10-18.

Hash Join

The optimizer usually uses a hash join when at least one of the two join tables does not have an index on the join column or when the database server must read a large number of rows from both tables. No index and no sorting is required when the database server performs a hash join.

A hash join consists of two activities: building the hash table (*build* phase) and probing the hash table (*probe* phase). Figure 10-2 shows the hash join in detail.

In the build phase, the database server reads one table and, after it applies any filters, creates a hash table. Think of a hash table conceptually as a series of *buckets*, each with an address that is derived from the key value by applying a hash function. The database server does not sort keys in a particular hash bucket.

Smaller hash tables can fit in the virtual portion of database server shared memory. The database server stores larger hash files on disk in the dbspace specified by the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable.

In the probe phase, the database server reads the other table in the join and applies any filters. For each row that satisfies the filters on the table, the database server applies the hash function on the key and probes the hash table to find a match.

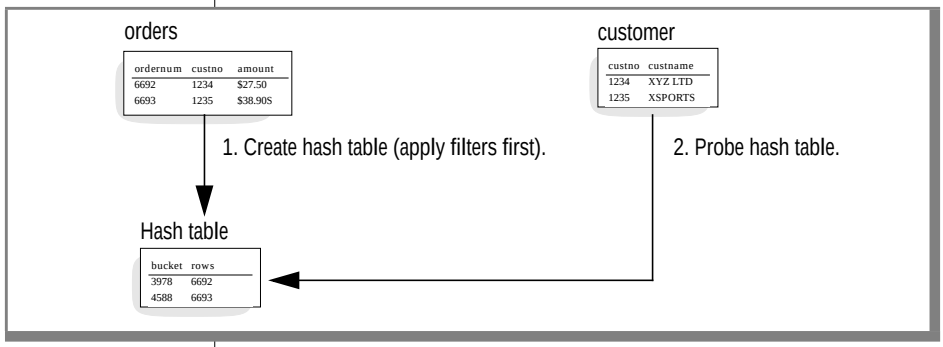


Figure 10-2
How a Hash Join Is Executed

Join Order

The order that tables are joined in a query is extremely important. A poor join order can cause query performance to decline noticeably.

The following SELECT statement calls for a three-way join:

```
SELECT C.customer_num, O.order_num
   FROM customer C, orders O, items I
   WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
```

The optimizer can choose one of the following join orders:

- Join **customer** to **orders**. Join the result to **items**.
- Join **orders** to **customer**. Join the result to **items**.
- Join **customer** to **items**. Join the result to **orders**.
- Join **items** to **customer**. Join the result to **orders**.
- Join **orders** to **items**. Join the result to **customer**.
- Join **items** to **orders**. Join the result to **customer**.

For an example of how the database server executes a plan according to a specific join order, refer to "Example of Query-Plan Execution" on page 10-8.

Example of Query-Plan Execution

The following SELECT statement calls for a three-way join:

```
SELECT C.customer_num, O.order_num
   FROM customer C, orders O, items I
   WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
```

Assume also that no indexes are on any of the three tables. Suppose that the optimizer chooses the **customer-orders-items** path and the nested-loop join for both joins (in reality, the optimizer usually chooses a hash join for two tables without indexes on the join columns). Figure 10-3 shows the *query plan*, expressed in pseudocode. For information about interpreting query plan information, see "Query Plan Report" on page 10-12.

```
for each row in the customer table do:
   read the row into C
   for each row in the orders table do:
      read the row into O
      if O.customer_num = C.customer_num then
         for each row in the items table do:
            read the row into I
            if I.order_num = O.order_num then
               accept the row and send to user
            end if
         end for
      end if
   end for
end for
```

Figure 10-3
A Query Plan in Pseudocode

This procedure reads the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once for each row of the **customer** table
- All rows of the **items** table once for each row of the **customer-orders** pair

This example does not describe the only possible query plan. Another plan merely reverses the roles of **customer** and **orders**: for each row of **orders**, it reads all rows of **customer**, looking for a matching **customer_num**. It reads the same number of rows in a different order and produces the same set of rows in a different order. In this example, no difference exists in the amount of work that the two possible query plans need to do.

Join with Column Filters

The presence of a *column filter* changes things. A column filter is a WHERE expression that reduces the number of rows that a table contributes to a join. The following example shows the preceding query with a filter added:

```
SELECT C.customer_num, O.order_num
   FROM customer C, orders O, items I
   WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
      AND O.paid_date IS NULL
```

The expression `O.paid_date IS NULL` filters out some rows, reducing the number of rows that are used from the **orders** table. Consider a plan that starts by reading from **orders**. Figure 10-4 displays this sample plan in pseudocode.

```
for each row in the orders table do:
   read the row into O
   if O.paid_date is null then
      for each row in the customer table do:
         read the row into C
         if O.customer_num = C.customer_num then
            for each row in the items table do:
               read the row into I
               if I.order_num = O.order_num then
                  accept row and return to user
               end if
            end for
         end if
      end for
   end if
end for
```

Figure 10-4
Query Plan That Uses a Column Filter

Let *pdnull* represent the number of rows in **orders** that pass the filter. It is the value of **COUNT(*)** that results from the following query:

```
SELECT COUNT(*) FROM orders WHERE paid_date IS NULL
```

If one customer exists for every order, the plan in Figure 10-4 reads the following rows:

- All rows of the **orders** table once
- All rows of the **customer** table, *pdnull* times
- All rows of the **items** table, *pdnull* times

Figure 10-5 shows an alternative execution plan that reads from the **customer** table first.

```
for each row in the customer table do:
   read the row into C
   for each row in the orders table do:
      read the row into O
      if O.paid_date is null and
         O.customer_num = C.customer_num then
         for each row in the items table do:
            read the row into I
            if I.order_num = O.order_num then
               accept row and return to user
            end if
         end for
      end if
   end for
```

Figure 10-5
The Alternative Query Plan in Pseudocode

Because the filter is not applied in the first step that Figure 10-5 shows, this plan reads the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once for every row of **customer**
- All rows of the **items** table, *pdnull* times

The query plans in Figure 10-4 and Figure 10-5 produce the same output in a different sequence. They differ in that one reads a table *pdnull* times, and the other reads a table SELECT COUNT(*) FROM customer times. By choosing the appropriate plan, the optimizer can save thousands of disk accesses in a real application.

Join with Indexes

The preceding examples do not use indexes or constraints. The presence of indexes and constraints provides the optimizer with options that can greatly improve query-execution time. Figure 10-6 shows the outline of a query plan for the previous query as it might be constructed using indexes.

```
for each row in the customer table do:
   read the row into C
   look up C.customer_num in index on orders.customer_num
   for each matching row in the orders index do:
      read the table row for O
      if O.paid_date is null then
         look up O.order_num in index on items.order_num
         for each matching row in the items index do:
            read the row for I
            construct output row and return to user
         end for
      end if
   end for
end for
```

Figure 10-6
Query Plan with Indexes

The keys in an index are sorted so that when the database server finds the first matching entry, it can read any other rows with identical keys without further searching, because they are located in physically adjacent positions. This query plan reads only the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once (because each order is associated with only one customer)
- Only rows in the **items** table that match *pdnull* rows from the **customer-orders** pairs

This query plan achieves a great reduction in cost compared with plans that do not use indexes. An inverse plan, reading **orders** first and looking up rows in the **customer** table by its index, is also feasible by the same reasoning.

The physical order of rows in a table also affects the cost of index use. To the degree that a table is ordered relative to an index, the overhead of accessing multiple table rows in index order is reduced. For example, if the **orders** table rows are physically ordered according to the customer number, multiple retrievals of orders for a given customer would proceed more rapidly than if the table were ordered randomly.

In some cases, using an index might incur additional costs. For more information, refer to “Index Lookup Costs” on page 10-32.

Query Plan Evaluation

The optimizer considers all query plans by analyzing factors such as disk I/O and CPU costs. It constructs all feasible plans simultaneously using a bottom-up, breadth-first search strategy. That is, the optimizer first constructs all possible join pairs. It eliminates the more expensive of any *redundant* pair, which are join pairs that contain the same tables and produce the same set of rows as another join pair. For example, if neither join specifies an ordered set of rows by using the ORDER BY or GROUP BY clauses of the SELECT statement, the join pair (A x B) is redundant with respect to (B x A).

If the query uses additional tables, the optimizer joins each remaining pair to a new table to form all possible join triplets, eliminating the more expensive of redundant triplets and so on for each additional table to be joined. When a nonredundant set of possible join combinations has been generated, the optimizer selects the plan that appears to have the lowest execution cost.

Query Plan Report

Any user who runs a query can use the SET EXPLAIN statement or the EXPLAIN directive to display the query plan that the optimizer chooses. For information on how to specify the directives, refer to "EXPLAIN Directives" on page 11-16. The user enters the SET EXPLAIN ON statement or the SET EXPLAIN ON AVOID_EXECUTE statement before the SQL statement for the query, as the following example shows.

```
SET EXPLAIN ON AVOID_EXECUTE;
SELECT * FROM customer, orders
WHERE customer.customer_num = orders.customer_num
   AND customer.lname = "Higgins";
```

After the database server executes the SET EXPLAIN ON statement, it writes an explanation of each query plan to a file for subsequent queries that the user enters. For a description of the output, refer to "EXPLAIN Output Description" on page 10-13.

EXPLAIN Output File

UNIX

On UNIX, the database server writes the output of the SET EXPLAIN ON statement or the EXPLAIN directive to the **sqexplain.out** file.

If the client application and the database server are on the same computer, the **sqexplain.out** file is stored in the current directory. If you are using a Version 5.x or earlier client application and the **sqexplain.out** file does not appear in the current directory, check your home directory for the file.

When the current database is on another computer, the **sqexplain.out** file is stored in your home directory on the remote host. ♦

WIN NT

On Windows NT, the database server writes the output of the SET EXPLAIN ON statement or the EXPLAIN directive to the file **%INFORMIXDIR%\sqexpln*username*.out**. ♦

EXPLAIN Output Description

The SET EXPLAIN output contains the following information:

- The SELECT statement for the query
- An estimate of the query cost in units the optimizer uses to compare plans

  These units represent a relative time for query execution, with each unit assumed to be roughly equivalent to a typical disk access. The optimizer chose this query plan because the estimated cost for its execution was the lowest among all the evaluated plans.
- An estimate for the number of rows that the query is expected to produce
- The order to access the tables during execution

- The access plan by which the database server reads each table

  The following table shows the possible access plans.

Access Plan	Effect
SEQUENTIAL SCAN	Reads rows in sequence
INDEX PATH	Scans one or more indexes
AUTOINDEX PATH	Creates a temporary index
REMOTE PATH	Accesses another database (distributed query)

- The table column or columns that serve as a filter, if any, and whether the filtering occurs through an index
- The join plan for each pair of tables

  The following table shows the possible join plans.

Join Plan	Effect
DYNAMIC HASH	Use a hash join on the preceding join-table pair. The output includes a list of the filters used to join the tables. If DYNAMIC HASH JOIN is followed by (Build Outer) in the output, the build phase occurs on the first table. Otherwise, the build occurs on the second table, preceding the DYNAMIC HASH JOIN.
NESTED LOOP	Use a hash join on the preceding join-table pair. The output includes a list of the filters used to join the tables. The optimizer lists the outer table first for each join pair.

Sample Query Plan Reports

The following sections describe sample query plans that you might want to display when analyzing the performance of queries.

Single-Table Query

Figure 10-7 shows the SET EXPLAIN output for a simple query.

```
QUERY:
------
SELECT fname, lname, company FROM customer

Estimated Cost: 2
Estimated # of Rows Returned: 28

  1) virginia.customer: SEQUENTIAL SCAN
```

Figure 10-7
SET EXPLAIN Output for a Simple Query

Figure 10-8 shows the SET EXPLAIN output for a complex query on the **customer** table.

Figure 10-8
SET EXPLAIN Output for a Complex Query

```
QUERY:
------
SELECT fname, lname, company FROM customer
WHERE company MATCHES 'Sport*' AND
    customer_num BETWEEN 110 AND 115
ORDER BY lname

Estimated Cost: 1
Estimated # of Rows Returned: 1
Temporary Files Required For: Order By

  1) virginia.customer: INDEX PATH

       Filters: virginia.customer.company MATCHES 'Sport*'

    (1) Index Keys: customer_num   (Serial, fragments: ALL)
        Lower Index Filter: virginia.customer.customer_num >= 110
        Upper Index Filter: virginia.customer.customer_num <= 115
```

The following output lines in Figure 10-8 show the scope of the index scan for the second query:

- Lower Index Filter: virginia.customer.customer_num >= 110

  Start the index scan with the index key value of `110`.
- Upper Index Filter: virginia.customer.customer_num <= 115

  Stop the index scan with the index key value of `115`.

Multitable Query

Figure 10-9 shows the SET EXPLAIN output for a multiple-table query.

Figure 10-9
SET EXPLAIN Output for a Multi-table Query

```
QUERY:
------
SELECT C.customer_num, O.order_num, SUM (I.total_price)
FROM customer C, orders O, items I
WHERE C.customer_num = O.customer_num
AND O.order_num = I.order_num
GROUP BY C.customer_num, O.order_num

Estimated Cost: 78
Estimated # of Rows Returned: 1
Temporary Files Required For: Group By

  1) virginia.o: SEQUENTIAL SCAN

  2) virginia.c: INDEX PATH

    (1) Index Keys: customer_num   (Key-Only)  (Serial, fragments: ALL)
        Lower Index Filter: virginia.c.customer_num = virginia.o.customer_num
NESTED LOOP JOIN

  3) virginia.i: INDEX PATH

    (1) Index Keys: order_num   (Serial, fragments: ALL)
        Lower Index Filter: virginia.o.order_num = virginia.i.order_num
NESTED LOOP JOIN
```

The SET EXPLAIN output lists the order in which the database server accesses the tables and the access plan to read each table. The plan in Figure 10-9 indicates that the database server is to perform the following actions:

1. The database server is to read the **orders** table first.

   Because no filter exists on the **orders** table, the database server must read all rows. Reading the table in physical order is the least expensive approach.

2. For each row of **orders**, the database server is to search for matching rows in the **customer** table.

   The search uses the index on **customer_num**. The notation `Key-Only` means that only the index need be read for the **customer** table because only the **c.customer_num** column is used in the join and the output, and the column is an index key.

3. For each row of **orders** that has a matching **customer_num**, the database server is to search for a match in the **items** table using the index on **order_num**.

Key-First Scan

A *key-first scan* is an index scan that uses keys other than those listed as lower and upper index filters. Figure 10-10 shows a sample query using a key-first scan.

Figure 10-10
SET EXPLAIN Output for a Key-First Scan

```
create index idx1 on tab1(c1, c2);

select * from tab1 where (c1 > 0) and ( (c2 = 1) or (c2 = 2))
Estimated Cost: 4
Estimated # of Rows Returned: 1

1) pubs.tab1: INDEX PATH

    Filters: (pubs.tab1.c2 = 1 OR pubs.tab1.c2 = 2)

    (1) Index Keys: c1 c2   (Key-First)  (Serial, fragments: ALL)
        Lower Index Filter: pubs.tab1.c1 > 0
```

Even though in this example the database server must eventually read the row data to return the query results, it attempts to reduce the number of possible rows by applying additional key filters first. The database server uses the index to apply the additional filter, `c2 = 1 OR c2 = 2`, before it reads the row data.

Query Plans for Subqueries

The optimizer can change a subquery to a join automatically if the join provides a lower cost. For example, Figure 10-11 sample output of the SET EXPLAIN ON statement shows that the optimizer changes the table in the subquery to be the inner table in a join.

```
QUERY:
------
SELECT company, fname, lname, phone
FROM customer c
WHERE EXISTS(
   SELECT customer_num FROM cust_calls u
      WHERE c.customer_num = u.customer_num)

Estimated Cost: 6
Estimated # of Rows Returned: 7

  1) virginia.c: SEQUENTIAL SCAN

  2) virginia.u: INDEX PATH  (First Row)

    (1) Index Keys: customer_num call_dtime   (Key-Only)  (Serial,
fragments: ALL)
        Lower Index Filter: virginia.c.customer_num =
virginia.u.customer_num
NESTED LOOP JOIN  (Semi Join)
```

Figure 10-11
SET EXPLAIN Output for a Flattened Subquery

For more information on the SET EXPLAIN ON statement, refer to "Query Plan Report" on page 10-12.

When the optimizer changes a subquery to a join, it can use several variations of the access plan and the join plan:

- First-row scan

  A first-row scan is a variation of a table scan. When the database server finds one match, the table scan halts.

- Skip-duplicate-index scan

  The skip-duplicate-index scan is a variation of an index scan. The database server does not scan duplicates.

- Semi join

  The semi join is a variation of a nested-loop join. The database server halts the inner-table scan when the first match is found. For more information on a semi join, refer to "Nested-Loop Join" on page 10-5.

Query Plans for Collection-Derived Tables

A collection-derived table is a special method that the database server uses to process a query on a collection. To use a collection-derived table, a query must contain the TABLE keyword in the FROM clause of an SQL statement. For more information about how to use collection-derived tables in an SQL statement, refer to the *Informix Guide to SQL: Syntax*.

Although the database does not actually create a table for the collection, it processes the data as if it were a table. Collection-derived tables allow developers to use fewer cursors and host variables to access a collection, in some cases.

These SQL statements create a collection column called **children**:

```
CREATE ROW TYPE person(name CHAR(255), id INT);
CREATE TABLE parents(name CHAR(255),
id INT,
children LIST(person NOT NULL));
```

The following query creates a collection-derived table for the **children** column and treats the elements of this collection as rows in a table:

```
SELECT name, id
FROM TABLE((SELECT children
FROM parents
WHERE parents.id = 1001)) c_table(name, id);
```

To complete this query, the database server performs the following steps:

1. Scans the **parent** table to find the row where `parents.id = 1001`

   This operation is listed as a SEQUENTIAL SCAN in the SET EXPLAIN output that Figure 10-12 shows.

2. Reads the value of the children **collection** column

3. Scans the single collection and returns the value of **name** and **id** to the application

   This operation is listed as a COLLECTION SCAN in the SET EXPLAIN output that Figure 10-12 shows.

```
QUERY:
------
select name, id
from table((select children
from parents
where parents.id = 1001))
c_table(name, id)

Estimated Cost: 2
Estimated # of Rows Returned: 1

  1) lsuto.c_table: COLLECTION SCAN
    Subquery:
    ---------
    Estimated Cost: 1
    Estimated # of Rows Returned: 1

      1) lsuto.parents: SEQUENTIAL SCAN

            Filters: lsuto.parents.id = 1001
```

Figure 10-12
Query Plan That Uses a Collection-Derived Table

Factors That Affect the Query Plan

When the optimizer determines the query plan, it assigns a cost to each possible plan and then chooses the plan with the lowest cost. Some of the factors that the optimizer uses to determine the cost of each query plan are as follows:

- The number of I/O requests that are associated with each filesystem access
- The CPU work that is required to determine which rows meet the query predicate
- The resources that are required to sort or group the data
- The amount of memory available for the query (specified by the DS_TOTAL_MEMORY and DS_MAX_QUERIES parameters)

To calculate the cost of each possible query plan, the optimizer:

- Uses a set of statistics that describes the nature and physical characteristics of the table data and indexes
- Examines the query filters
- Examines the indexes that could be used in the plan
- Uses the cost of moving data to perform joins locally or remotely for distributed queries

Statistics Held for the Table and Index

The accuracy with which the optimizer can assess the execution cost of a query plan depends on the information available to the optimizer. Use the UPDATE STATISTICS statement to maintain simple statistics about a table and its associated indexes. Updated statistics provide the query optimizer with information that can minimize the amount of time required to perform queries on that table.

The database server initializes a statistical profile of a table when the table is created, and the profile is refreshed when you issue the UPDATE STATISTICS statement. The query optimizer does not recalculate the profile for tables automatically. In some cases, gathering the statistics might take longer than executing the query.

To ensure that the optimizer selects a query plan that best reflects the current state of your tables, run UPDATE STATISTICS at regular intervals. For guidelines, refer to "Updating Statistics" on page 13-13.

The optimizer uses the following system catalog information as it creates a query plan:

- The number of rows in a table, as of the most recent UPDATE STATISTICS statement
- Whether a column is constrained to be unique
- The distribution of column values, when requested with the MEDIUM or HIGH keyword in the UPDATE STATISTICS statement

  For more information on data distributions, refer to "Creating Data Distributions" on page 13-15.
- The number of disk pages that contain row data

The optimizer also uses the following system catalog information about indexes:

- The indexes that exist on a table, including the columns that they index, whether they are ascending or descending, and whether they are clustered
- The depth of the index structure (a measure of the amount of work that is needed to perform an index lookup)
- The number of disk pages that index entries occupy
- The number of unique entries in an index, which can be used to estimate the number of rows that an equality filter returns
- Second-largest and second-smallest key values in an indexed column

Only the second-largest and second-smallest key values are noted, because the extreme values might have a special meaning that is not related to the rest of the data in the column. The database server assumes that key values are distributed evenly between the second largest and second smallest. Only the initial 4 bytes of these keys are stored. If you create a distribution for a column associated with an index, the optimizer uses that distribution when it estimates the number of rows that match a query.

For more information on system catalog tables, refer to the *Informix Guide to SQL: Reference*.

Filters in the Query

The optimizer bases query-cost estimates on the number of rows to be retrieved from each table. In turn, the estimated number of rows is based on the *selectivity* of each conditional expression that is used within the WHERE clause. A conditional expression that is used to select rows is termed a *filter*.

The selectivity is a value between 0 and 1 that indicates the proportion of rows within the table that the filter can pass. A selective filter, one that passes few rows, has a selectivity near 0, and a filter that passes almost all rows has a selectivity near 1. For guidelines on filters, see "Improving Filter Selectivity" on page 13-7.

The optimizer can use data distributions to calculate selectivities for the filters in a query. However, in the absence of data distributions, the database server calculates selectivities for filters of different types based on table indexes. The following table lists some of the selectivities that the optimizer assigns to filters of different types. Selectivities calculated using data distributions are even more accurate than the ones that this table shows.

Filter Expression	Selectivity (F)
indexed-col = *literal-value* *indexed-col* = *host-variable* *indexed-col* IS NULL	F = 1 / (number of distinct keys in index)
tab1.indexed-col = *tab2.indexed-col*	F = 1 / (number of distinct keys in the larger index)
indexed-col > *literal-value*	F = (2nd-max - *literal-value*) / (2nd-max - 2nd-min)
indexed-col < *literal-value*	F = (*literal-value* - 2nd-min) / (2nd-max - 2nd-min)
any-col IS NULL *any-col* = *any-expression*	F = 1 / 10
any-col > *any-expression* *any-col* < *any-expression*	F = 1 / 3
any-col MATCHES *any-expression* *any-col* LIKE *any-expression*	F = 1 / 5

(1 of 2)

Filter Expression	Selectivity (F)
EXISTS *subquery*	F = 1 if *subquery* estimated to return >0 rows, else 0
NOT *expression*	F = 1 - F(*expression*)
expr1 AND *expr2*	F = F(*expr1*) ¥ F(*expr2*)
expr1 OR *expr2*	F = F(*expr1*) + F(*expr2*) - (F(*expr1*) ¥ F(*expr2*))
any-col IN *list*	Treated as any-col = $item_1$ ORºOR *any-col* = $item_{n.}$
any-col relop ANY *subquery*	Treated as any-col relop $value_1$ OR º OR *any-col relop* $value_n$ for estimated size of subquery *n*. *relop* is any relational operator, such as <, >, >=, <=.

Key:

indexed-col: first or only column in an index

2nd-max, *2nd-min*: second-largest and second-smallest key values in indexed column

(2 of 2)

Indexes for Evaluating a Filter

The optimizer notes whether an index can be used to evaluate a filter. For this purpose, an indexed column is any single column with an index or the first column named in a composite index. If the values contained in the index are all that is required, the rows are not read. It is faster to omit the page lookups for data pages whenever the database server can read values directly from the index.

The optimizer can choose an index for any one of the following cases:

- When the column is indexed and a value to be compared is a literal, a host variable, or an uncorrelated subquery

  The database server can locate relevant rows in the table by first finding the row in an appropriate index. If an appropriate index is not available, the database server must scan each table in its entirety.

- When the column is indexed and the value to be compared is a column in another table (a join expression)

  The database server can use the index to find matching values. The following join expression shows such an example:

  ```
  WHERE customer.customer_num = orders.customer_num
  ```

  If rows of **customer** are read first, values of **customer_num** can be applied to an index on **orders.customer_num**.

- When processing an ORDER BY clause

  If all the columns in the clause appear in the required sequence within a single index, the database server can use the index to read the rows in their ordered sequence, thus avoiding a sort.

- When processing a GROUP BY clause

  If all the columns in the clause appear in one index, the database server can read groups with equal keys from the index without requiring additional processing after the rows are retrieved from their tables.

Effect of PDQ on the Query Plan

When the parallel database query (PDQ) feature is turned on, the optimizer can choose to execute a query in parallel, which can improve performance dramatically when the database server processes queries that decision-support applications initiate. For more information, refer to Chapter 12, "Parallel Database Query."

Effect of OPTCOMPIND on the Query Plan

The OPTCOMPIND setting influences the access plan that the optimizer chooses for single and multitable queries, as the following sections describe.

Single-Table Query

For single-table scans, when OPTCOMPIND is set to `0` or `1` and the current transaction isolation level is Repeatable Read, the optimizer considers the following access plans:

- If an index is available, the optimizer uses it to access the table.
- If no index is available, the optimizer considers scanning the table in physical order.

When OPTCOMPIND is not set in the database server configuration, its value defaults to `2`. When OPTCOMPIND is set to `2` or `1` and the current isolation level is not Repeatable Read, the optimizer chooses the least expensive plan to access the table.

Multitable Query

For join plans, the OPTCOMPIND setting influences the access plan for a specific ordered pair of tables. Set OPTCOMPIND to `0` if you want the database server to select a join method exactly as it did in previous versions. This option ensures compatibility with previous versions of the database server.

If OPTCOMPIND is set to `0` or set to `1` and the current transaction isolation level is Repeatable Read, the optimizer gives preference to the nested-loop join.

Important: *When OPTCOMPIND is set to* `0`*, the optimizer does not choose a hash join*.

If OPTCOMPIND is set to `2` or set to `1` and the transaction isolation level is not Repeatable Read, the optimizer chooses the least expensive query plan from among those previously listed and gives no preference to the nested-loop join.

Effect of Available Memory on the Query Plan

The database server constrains the amount of memory that a parallel query can use based on the values of the DS_TOTAL_MEMORY and DS_MAX_QUERIES parameters. If the amount of memory available for the query is too low to execute a hash join, the database server uses a nested-loop join instead.

For more information on parallel queries and the DS_TOTAL_MEMORY and DS_MAX_QUERIES parameters, refer to Chapter 12, "Parallel Database Query."

Time Costs of a Query

This section explains the response-time effects of actions that the database server performs as it processes a query.

Many of the costs described cannot be reduced by adjusting the construction of the query. A few can be, however. The following costs can be reduced by optimal query construction and appropriate indexes:

- Sort time
- Data mismatches
- In-place ALTER TABLE
- Index lookups

For information about how to optimize specific queries, see Chapter 13, "Improving Individual Query Performance."

Memory-Activity Costs

The database server can process only data in memory. It must read rows into memory to evaluate those rows against the filters of a query. Once the database server finds rows that satisfy those filters, it prepares an output row in memory by assembling the selected columns.

Most of these activities are performed quickly. Depending on the computer and its workload, the database server can perform hundreds or even thousands of comparisons each second. As a result, the time spent on in-memory work is usually a small part of the execution time.

Although some in-memory activities, such as sorting, take a significant amount of time, it takes much longer to read a row from disk than to examine a row that is already in memory.

Sort-Time Costs

A sort requires in-memory work as well as disk work. The in-memory work depends on the number of columns that are sorted, the width of the combined sort key, and the number of row combinations that pass the query filter. Use the following formula to calculate the in-memory work that a sort operation requires:

$$W_m = (c * N_{fr}) + (w * N_{fr} \log_2(N_{fr}))$$

W_m is the in-memory work.

c is the number of columns to order and represents the costs to extract column values from the row and concatenate them into a sort key.

w is proportional to the width of the combined sort key in bytes and stands for the work to copy or compare one sort key. A numeric value for w depends strongly on the computer hardware in use.

N_{fr} is the number of rows that pass the query filter.

Sorting can involve writing information temporarily to disk if the amount of data to sort is large. You can direct the disk writes to occur in the operating-system file space or in a dbspace that the database server manages. For details, refer to "Configuring Dbspaces for Temporary Tables and Sort Files" on page 5-12.

The disk work depends on the number of disk pages where rows appear, the number of rows that meet the conditions of the query predicate, the number of rows that can be placed on a sorted page, and the number of merge operations that must be performed. Use the following formula to calculate the disk work that a sort operation requires:

$$W_d = p + (N_{fr}/N_{rp}) * 2 * (m - 1))$$

W_d is the disk work.

p is the number of disk pages.

N_{fr} is the number of rows that pass the filters.

N_{rp} is the number of rows that can be placed on a page.

m represents the number of *levels of merge* that the sort must use.

The factor m depends on the number of sort keys that can be held in memory. If there are no filters, N_{fr}/N_{rp} is equivalent to p.

When all the keys can be held in memory, m=1 and the disk work is equivalent to p. In other words, the rows are read and sorted in memory.

For moderate to large tables, rows are sorted in batches that fit in memory, and then the batches are merged. When m=2, the rows are read, sorted, and written in batches. Then the batches are read again and merged, resulting in disk work proportional to the following value:

$$W_d = +p(2 * (N_{fr}/N_{rp}))$$

The more specific the filters, the fewer the rows that are sorted. As the number of rows increases, and the amount of memory decreases, the amount of disk work increases.

To reduce the cost of sorting, use the following methods:

- Make your filters as specific (selective) as possible.
- Limit the projection list to the columns that are relevant to your problem.

Row-Reading Costs

When the database server needs to examine a row that is not already in memory, it must read that row from disk. The database server does not read only one row; it reads the entire page that contains the row. If the row spans more than one page, it reads all of the pages.

The actual cost of reading a page is variable and hard to predict. It is a combination of the following factors.

Factor	Effect of Factor
Buffering	If the needed page is in a page buffer already, the cost to read is nearly zero.
Contention	If two or more applications require access to the disk hardware, I/O requests can be delayed.
Seek time	The slowest thing that a disk does is to *seek*; that is, to move the access arm to the track that holds the data. Seek time depends on the speed of the disk and the location of the disk arm when the operation starts. Seek time varies from zero to nearly a second.
Latency	The transfer cannot start until the beginning of the page rotates under the access arm. This *latency*, or rotational delay, depends on the speed of the disk and on the position of the disk when the operation starts. Latency can vary from zero to a few milliseconds.

The time cost of reading a page can vary from microseconds for a page that is already in a buffer, to a few milliseconds when contention is zero and the disk arm is already in position, to hundreds of milliseconds when the page is in contention and the disk arm is over a distant cylinder of the disk.

Sequential Access Costs

Disk costs are lowest when the database server reads the rows of a table in physical order. When the first row on a page is requested, the disk page is read into a buffer page. Once the page is read in, it need not be read again; requests for subsequent rows on that page are filled from the buffer until all the rows on that page are processed. When one page is exhausted, the page for the next set of rows must be read in. To make sure that the next page is ready in memory, use the read-ahead configuration parameters described in "RA_PAGES and RA_THRESHOLD" on page 5-40.

When you use unbuffered devices for dbspaces, and the table is organized properly, the disk pages of consecutive rows are placed in consecutive locations on the disk. This arrangement allows the access arm to move very little when it reads sequentially. In addition, latency costs are usually lower when pages are read sequentially.

Nonsequential Access Costs

Whenever a table is read in random order, additional disk accesses are required to read the rows in the required order. Disk costs are higher when the rows of a table are read in a sequence unrelated to physical order on disk. Because the pages are not read sequentially from the disk, both seek and rotational delays occur before each page can be read. As a result, the disk-access time is much higher when a disk device reads table pages nonsequentially than when it reads that same table sequentially.

Nonsequential access often occurs when you use an index to locate rows. Although index entries are sequential, there is no guarantee that rows with adjacent index entries must reside on the same (or adjacent) data pages. In many cases, a separate disk access must be made to fetch the page for each row located through an index. If a table is larger than the page buffers, a page that contained a row previously read might be cleaned (removed from the buffer and written back to the disk) before a subsequent request for another row on that page can be processed. That page might have to be read in again.

Depending on the relative ordering of the table with respect to the index, you can sometimes retrieve pages that contain several needed rows. The degree to which the physical ordering of rows on disk corresponds to the order of entries in the index is called *clustering*. A highly clustered table is one in which the physical ordering on disk corresponds closely to the index.

Index Lookup Costs

The database server incurs additional costs when it finds a row through an index. The index is stored on disk, and its pages must be read into memory with the data pages that contain the desired rows.

An index lookup works down from the root page to a leaf page. The root page, because it is used so often, is almost always found in a page buffer. The odds of finding a leaf page in a buffer depend on the size of the index, the form of the query, and the frequency of column-value duplication. If each value occurs only once in the index and the query is a join, each row to be joined requires a nonsequential lookup into the index, followed by a nonsequential access to the associated row in the table.

Reading Duplicate Values From an Index

Using an index incurs an additional cost for duplicate values over reading the table sequentially. Each entry or set of entries with the same value must be located in the index. Then, for each entry in the index, a random access must be made to the table to read the associated row. However, if there are many duplicate rows per distinct index value, and the associated table is highly clustered, the added cost of joining through the index can be slight.

Searching for NCHAR or NVARCHAR Columns in an Index

GLS

Indexes that are built on NCHAR or NVARCHAR columns are sorted using a locale-specific comparison value. For example, the Spanish double-l character (ll) might be treated as a single unique character instead of a pair of ls.

In some locales, the comparison value is not based on the code-set order. The index build uses the locale-specific comparison value to store the key values in the index. As a result, a query using an index on an NCHAR or NVARCHAR scans the entire index because the database server searches the index in code-set order. ♦

In-Place ALTER TABLE Costs

For certain conditions, the database server uses an in-place alter algorithm to modify each row when you execute an ALTER TABLE statement (rather than during the alter table operation). After the alter table operation, the database server inserts rows using the latest definition.

If your query accesses rows that are not yet converted to the new table definition, you might notice a slight degradation in the performance of your individual query because the database server reformats each row in memory before it is returned.

For more information on the conditions and performance advantages when an in-place alter occurs, refer to "Altering a Table Definition" on page 6-51.

View Costs

You can create views of tables for a number of reasons:

- To limit the data that a user can access
- To reduce the time that it takes to write a complex query
- To hide the complexity of the query that a user needs to write

However, a query against a view might execute more slowly than expected when the complexity of the view definition causes a temporary table to be created to process the query. This temporary table is referred to as a *materialized view*. For example, you can create a view with a union to combine results from several SELECT statements.

The following sample SQL statement creates a view that includes unions:

```
CREATE VIEW view1 (col1, col2, col3, col4)
   AS
      SELECT a, b, c, d
         FROM tab1 WHERE ...
      UNION
      SELECT a2, b2, c2, d2
         FROM tab2 WHERE º
...
      UNION
      SELECT an, bn, cn, dn
         FROM tabn WHERE º
;
```

When you create a view that contains complex SELECT statements, the end user does not need to handle the complexity. The end user can just write a simple query, as the following example shows:

```
SELECT a, b, c, d
   FROM view1
      WHERE a < 10;
```

However, this query against **view1** might execute more slowly than expected because the database server creates a fragmented temporary table for the view before it executes the query.

Another situation when the query might execute more slowly than expected is if you use a view in an ANSI join. The complexity of the view definition might cause a temporary table to be created.

To determine if you have a query that must build a temporary table to process the view, execute the SET EXPLAIN statement. If you see `Temp Table For View` in the SET EXPLAIN output file, your query requires a temporary table to process the view.

Small-Table Costs

A table is small if it occupies so few pages that it can be retained entirely in the page buffers. Operations on small tables are generally faster than operations on large tables.

As an example, in the **stores_demo** database, the **state** table that relates abbreviations to names of states has a total size of fewer than 1000 bytes; it fits in no more than two pages. This table can be included in any query at little cost. No matter how this table is used, it costs no more than two disk accesses to retrieve this table from disk the first time that it is required.

Data-Mismatch Costs

An SQL statement can encounter additional costs when the data type of a column that is used in a condition differs from the definition of the column in the CREATE TABLE statement.

For example, the following query contains a condition that compares a column to a data type value that differs from the table definition:

```
CREATE TABLE table1 (a integer, ...);
SELECT * FROM table1
   WHERE a = '123';
```

The database server rewrites this query before execution to convert 123 to an integer. The SET EXPLAIN output shows the query in its adjusted format. This data conversion has no noticeable overhead.

The additional costs of a data mismatch are most severe when the query compares a character column with a noncharacter value and the length of the number is not equal to the length of the character column. For example, the following query contains a condition in the WHERE clause that equates a character column to an integer value because of missing quotation marks:

```
CREATE TABLE table2 (char_col char(3), ...);
SELECT * FROM table2
   WHERE char_col = 1;
```

This query finds all of the following values that are valid for **char_col**:

```
' 1'
'001'
'1'
```

These values are not necessarily clustered together in the index keys. Therefore, the index does not provide a fast and correct way to obtain the data. The SET EXPLAIN output shows a sequential scan for this situation.

Warning: *The database server does not use an index when the SQL statement compares a character column with a noncharacter value that is not equal in length to the character column.*

GLS

GLS Functionality Costs

As section "Searching for NCHAR or NVARCHAR Columns in an Index" on page 10-32 describes, indexing certain data sets cause significant performance degradation. Sorting certain data sets can also degrade performance.

If you do not need a non-ASCII collation sequence, Informix recommends that you use the CHAR and VARCHAR data types for character columns whenever possible. Because CHAR and VARCHAR data require simple value-based comparison, sorting and indexing these columns is less expensive than for non-ASCII data types (NCHAR or NVARCHAR, for example). For more information on other character data types, see the *Informix Guide to GLS Functionality*.

Network-Access Costs

Moving data over a network imposes delays in addition to those you encounter with direct disk access. Network delays can occur when the application sends a query or update request across the network to a database server on another computer. Although the database server performs the query on the remote host computer, that database server returns the output to the application over the network.

Data sent over a network consists of command messages and buffer-sized blocks of row data. Although the details can differ depending on the network and the computers, the database server network activity follows a simple model in which one computer, the *client*, sends a request to another computer, the *server*. The server replies with a block of data from a table.

Whenever data is exchanged over a network, delays are inevitable in the following situations:

- When the network is busy, the client must wait its turn to transmit. Such delays are usually less than a millisecond. However, on a heavily loaded network, these delays can increase exponentially to tenths of seconds and more.
- When the server is handling requests from more than one client, requests might be queued for a time that can range from milliseconds to seconds.
- When the server acts on the request, it incurs the time costs of disk access and in-memory operations that the preceding sections describe.

Transmission of the response is also subject to network delays.

Network access time is extremely variable. In the best case, when neither the network nor the server is busy, transmission and queueing delays are insignificant, and the server sends a row almost as quickly as a local database server might. Furthermore, when the client asks for a second row, the page is likely to be in the page buffers of the server.

Unfortunately, as network load increases, all these factors tend to worsen at the same time. Transmission delays rise in both directions, which increases the queue at the server. The delay between requests decreases the likelihood of a page remaining in the page buffer of the responder. Thus, network-access costs can change suddenly and quite dramatically.

If you use the SELECT FIRST *n* clause in a distributed query, you will still see only the requested amount of data. However, the local database server does not send the SELECT FIRST *n* clause to the remote site. Therefore, the remote site might return more data.

The optimizer that the database server uses assumes that access to a row over the network takes longer than access to a row in a local database. This estimate includes the cost of retrieving the row from disk and transmitting it across the network.

For information on actions that might improve performance across the network, refer to the following sections:

- "Improving Performance of Distributed Queries" on page 13-26
- "MaxConnect for Multiple Connections" on page 3-34
- "Multiplexed Connections" on page 3-32
- "Network Buffer Pools" on page 3-22

SQL Within SPL Routines

The following sections contain information about how and when the database server optimizes and executes SQL within an SPL routine.

SQL Optimization

If an SPL routine contains SQL statements, at some point the query optimizer evaluates the possible query plans for SQL in the SPL routine and selects the query plan with the lowest cost. The database server puts the selected query plan for each SQL statement in an execution plan for the SPL routine.

When you create an SPL routine with the CREATE PROCEDURE statement, the database server attempts to optimize the SQL statements within the SPL routine at that time. If the tables cannot be examined at compile time (because they do not exist or are not available), the creation does not fail. In this case, the database server optimizes the SQL statements the first time that the SPL routine executes.

The database server stores the optimized execution plan in the **sysprocplan** system catalog table for use by other processes. In addition, the database server stores information about the SPL routine (such as procedure name and owner) in the sysprocedures system catalog table and an ASCII version of the SPL routine in the sysprocbody system catalog table.

Figure 10-13 summarizes the information that the database server stores in system catalog tables during the compilation process.

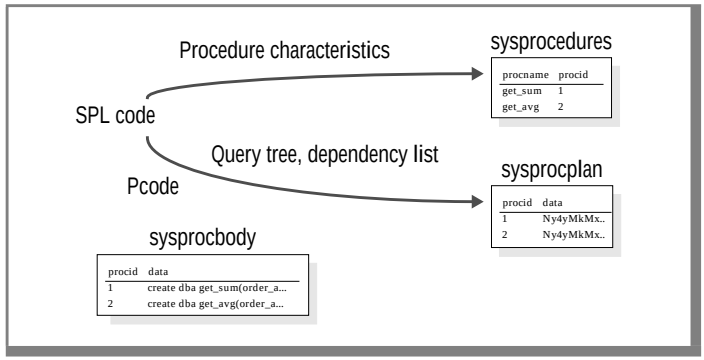


Figure 10-13
SPL Information Stored in System Catalog Tables

Displaying the Execution Plan

When you execute an SPL routine, it is already optimized. To display the query plan for each SQL statement contained in the SPL routine, execute the SET EXPLAIN ON statement prior to one of the following SQL statements that always tries to optimize the SPL routine:

- CREATE PROCEDURE
- UPDATE STATISTICS FOR PROCEDURE

  For example, use the following statements to display the query plan for an SPL routine:

```
SET EXPLAIN ON;
UPDATE STATISTICS FOR PROCEDURE procname;
```

Automatic Reoptimization

The database server uses the dependency list to keep track of changes that would cause reoptimization the next time that an SPL routine executes.

The database server reoptimizes an SQL statement the next time an SPL routine executes after one of the following situations:

- Execution of any data definition language (DDL) statement (such as ALTER TABLE, DROP INDEX, and CREATE INDEX) that might alter the query plan
- Alteration of a table that is linked to another table with a referential constraint (in either direction)
- Execution of UPDATE STATISTICS FOR TABLE for any table involved in the query

  The UPDATE STATISTICS FOR TABLE statement changes the version number of the specified table in **systables.**
- Renaming a column, database, or index with the RENAME statement

Whenever the SPL routine is reoptimized, the database server updates the **sysprocplan** system catalog table with the reoptimized execution plan.

Reoptimizing SPL Routines

If you do not want to incur the cost of automatic reoptimization when you first execute an SPL routine after one of the situations that "Automatic Reoptimization" on page 10-40 lists, execute the UPDATE STATISTICS statement with the FOR PROCEDURE clause immediately after the situation occurs. In this way, the SPL routine is reoptimized before any users execute it. To prevent unnecessary reoptimization of all SPL routines, ensure that you specify a specific procedure name in the FOR PROCEDURE clause.

```
UPDATE STATISTICS FOR PROCEDURE(myroutine);
```

For guidelines to run UPDATE STATISTICS, refer to "Updating Statistics" on page 13-13.

Optimization Levels for SQL in SPL Routines

The current optimization level set in an SPL routine affects how the SPL routine is optimized.

The algorithm that a SET OPTIMIZATION HIGH statement invokes is a sophisticated, cost-based strategy that examines all reasonable query plans and selects the best overall alternative. For large joins, this algorithm can incur more overhead than desired. In extreme cases, you can run out of memory.

The alternative algorithm that a SET OPTIMIZATION LOW statement invokes eliminates unlikely join strategies during the early stages, which reduces the time and resources spent during optimization. However, when you specify a low level of optimization, the optimal strategy might not be selected because it was eliminated from consideration during early stages of the algorithm.

For SPL routines that remain unchanged or change only slightly and that contain complex SELECT statements, you might want to set the SET OPTIMIZATION statement to `HIGH` when you create the SPL routine. This optimization level stores the best query plans for the SPL routine. Then set optimization to `LOW` before you execute the SPL routine. The SPL routine then uses the optimal query plans and runs at the more cost-effective rate if reoptimization occurs.

Execution of an SPL Routine

When the database server executes an SPL routine with the EXECUTE PROCEDURE statement, with the CALL statement, or within an SQL statement, the following activities occur:

- The database server reads the interpreter code from the system catalog tables and converts it from a compressed format to an executable format. If the SPL routine is in the UDR cache, the database server retrieves it from the cache and bypasses the conversion step.
- The database server executes any SPL statements that it encounters.

- When the database server encounters an SQL statement, it retrieves the query plan from the database and executes the statement. If the query plan has not been created, the database server optimizes the SQL statement before it executes.
- When the database server reaches the end of the SPL routine or when it encounters a RETURN statement, it returns any results to the client application. Unless the RETURN statement has a WITH RESUME clause, the SPL routine execution is complete.

UDR Cache

The first time that a user executes an SPL routine, the database server stores the executable format and any query plans in the UDR cache in the virtual portion of shared memory. When another user executes an SPL routine, the database server first checks the UDR cache. SPL execution performance improves when the database server can execute the SPL routine from the UDR cache. The UDR cache also stores UDRs, user-defined aggregates, and extended data types definitions.

Changing the UDR Cache

The default number of SPL routines, UDRs, and other user-defined definitions in the UDR cache is 127. You can change the number of entries with the PC_POOLSIZE CONFIGURATION parameter.

The database server uses a hashing algorithm to store and locate SPL routines in the UDR cache. You can modify the number of *buckets* in the UDR cache with the PC_HASHSIZE configuration parameter. For example, if PC_POOLSIZE is `100` and PC_HASHSIZE is `10`, each bucket can have up to 10 SPL routines and UDRs.

Too many buckets cause the database server to move out cached SPL routines when the bucket fills. Too few buckets increase the number of SPL routines in a bucket, and the database server must search though the SPL routines in a bucket to determine if the SPL routine that it needs is there.

Once the number of entries in a bucket reaches 75 percent, the database server removes the least recently used SPL routines from the bucket (and hence from the UDR cache) until the number of SPL routines in the bucket is 50 percent of the maximum SPL routines in the bucket.

Important: *PC_POOLSIZE and PC_HASHSIZE also control other memory caches for the database server (excluding the buffer pool, the SQL statement cache, the data distribution cache, and the data-dictionary cache). When you modify the size and number of hash buckets for SPL routines, you also modify the size and number of hash buckets for the other caches (such as the aggregate cache, oplcass, and typename cache).*

Monitoring the UDR Cache

Execute **onstat -g prc** to monitor the UDR cache. You can also execute **onstat -g cac** to list the contents of other memory caches (such as the aggregate cache) as well as the UDR cache.

Figure 10-14 shows sample output for **onstat -g prc**.

```
UDR Cache:
    Number of lists             : 31
    PC_POOLSIZE                 : 127
UDR Cache Entries:
list#  id  ref_cnt   dropped?   heap_ptr        udr name
--------------------------------------------------------------
0       138     0       0       a4ba820         sales_rep@london:.destroy
3       50      0       0       a4b2020         sales_rep@london:.assign
6       25      0       0       a4b8420         sales_rep@london:.rowoutput
7       29      0       0       a214860         sales_rep@london:.assign
...
```

Figure 10-14
onstat -g prc Output

The **onstat -g prc** output has the following fields.

Field	Description
Number of Lists	Number of buckets that PC_HASHSIZE specifies
PC_POOLSIZE	Number of UDRs and SPL routines allowed in the UDR cache
List #	Bucket number
ID	Unique ID of the UDR or SPL routines
Ref count	Number of times that users have executed the UDR or SPL routine from the cache
Dropped	Designation if the SPL routine has been marked to be dropped
Heap ptr	Heap pointer
UDR name	Name of the UDR or SPL routine

Trigger Execution

A trigger is a database mechanism that executes an SQL statement automatically when a certain event occurs. You can set up a trigger on a table to execute for a SELECT, INSERT, UPDATE or DELETE statement. For more information about using triggers, consult the *Informix Guide to SQL: Tutorial* and the CREATE TRIGGER statement in the *Informix Guide to SQL: Syntax*.

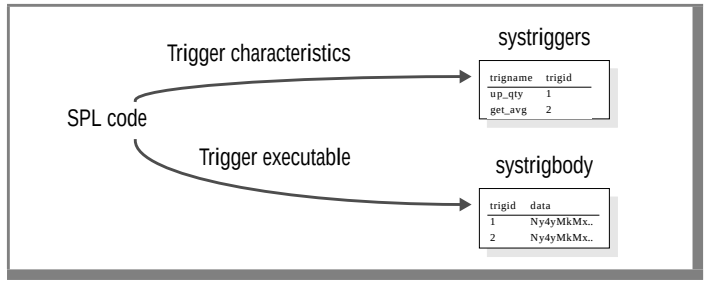


Figure 10-15
Trigger Information Stored in System Catalog Tables

When you create a trigger with the CREATE TRIGGER statement, the database server puts information about the trigger and the statement that it executes in two system catalog tables, **systriggers**, and **systrigbody**. In addition, dictionary tables stored in memory indicate whether the table has triggers defined for it. Whenever the database server executes a SELECT, INSERT, UPDATE, or DELETE statement, it must check if a trigger is defined for the type of statement and the table and columns on which the statement operates. If the statement requires a trigger to be executed, the database server retrieves the statement text from **systrigbody** and executes the SPL routine when required before, during, and after the statement.

Performance Implications for Triggers

Triggers can improve performance slightly because of the reduction in the number of messages passed from the client to the database server. For example, if the trigger fires five SQL statements, the client saves at least 10 messages passed between the client and database server (one to send the SQL statement and one for the reply after the database server executes the SQL statement). Triggers improve performance the most when they execute more SQL statements and the network speed is comparatively slow.

When the database server executes an SQL statement, it must perform the following actions:

- Determine if a trigger must be fired
- Retrieve the trigger from **systriggers** and **systrigbody**

These operations cause only a slight performance impact that can be offset by the decreased number of messages passed between the client and the server.

However, triggers executed on SELECT statements have additional performance implications. The following sections explain these implications.

SELECT Triggers on Tables in a Table Hierarchy

When the database server executes a SELECT statement that includes a table that is involved in a table hierarchy, and the SELECT statement fires a SELECT trigger, performance might be slower if the SELECT statement that invokes the trigger involves a join, sort, or materialized view. In this case, the database server does not know which columns are affected in the table hierarchy, so it can execute the query differently. The following behaviors might occur:

- Key-only index scans are disabled on the table that is involved in a table hierarchy.
- If the database server needs to sort data selected from a table involved in a table hierarchy, it copies all of the columns in the SELECT list to the temporary table, not just the sort columns.
- If the database server uses the table included in the table hierarchy to build a hash table for a hash join with another table, it bypasses the early projection, meaning it uses all of the columns from the table to build the hash table, not just the columns in the join.
- If the SELECT statement contains a materialized view (meaning a temporary table must be built for the columns in a view) that contains columns from a table involved in a table hierarchy, all columns from the table are included in the temporary table, not just the columns actually contained in the view.

SELECT Triggers and Row Buffering

In SELECT statements whose tables do not fire SELECT triggers, the database server sends more than one row back to the client and stores the rows in a buffer even though the client application requested only one row with a FETCH statement. However, for SELECT statements that contain one or more tables that fire a SELECT trigger, the database server sends only the requested row back to the client instead of a buffer full. The database server cannot return other rows to the client until the trigger action occurs.

The lack of buffering for SELECT statements that fire SELECT triggers might reduce performance slightly compared to an identical SELECT statement that does not fire a SELECT trigger.

Chapter

11

Optimizer Directives

In This Chapter 11-3

Reasons to Use Optimizer Directives. 11-3

Guidelines for Using Directives 11-5

Preparation for Using Directives 11-6

Types of Directives 11-7
- Access-Plan Directives 11-8
- Join-Order Directives 11-8
  - Effect of Join Order on Join Plan 11-8
  - Join Order When You Use Views 11-9
- Join-Plan Directives 11-10
- Optimization-Goal Directives 11-11
- Example with Directives 11-12
- EXPLAIN Directives 11-16

Configuration Parameters and Environment Variables for Directives 11-18

Directives and SPL Routines 11-18

In This Chapter

This chapter describes how to use directives to improve query performance.

Optimizer directives are comments in a SELECT statement that instruct the query optimizer how to execute a query. You can also place directives in UPDATE and DELETE statements, instructing the optimizer how to access the data. Optimizer directives can either be explicit directions (for example, use `this index` or `access this table first`), or they can eliminate possible query plans (for example, `do not read this table sequentially` or `do not perform a nested-loop join`).

Reasons to Use Optimizer Directives

You can use optimizer directives when the optimizer does not choose the best query plan to perform a query, because of the complexity of the query, or because it does not have enough information about the nature of the data. A poor query plan provides poor performance.

Before you decide when to use optimizer directives, you should understand what makes a good query plan.

Although the optimizer creates a query plan based on costs of using different table-access paths, join orders, and join plans, it generally chooses a query plan that follows these guidelines:

- Do not use an index when the database server must read a large portion of the table. For example, the following query might read most of the **customer** table:

  ```
  SELECT * FROM customer WHERE STATE <> "ALASKA";
  ```

  Assuming the customers are evenly spread among all 50 states, you might estimate that the database server must read 98 percent of the table. It is more efficient to read the table sequentially than to traverse an index (and subsequently the data pages) when the database server must read most of the rows.

- When you choose between indexes to access a table, use an index that can rule out the most rows. For example, consider the following query:

  ```
  SELECT * FROM customer
  WHERE state = "NEW YORK" AND order_date = "01/20/97"
  ```

  Assuming that 200,000 customers live in New York and only 1000 customers ordered on any one day, the optimizer most likely chooses an index on **order_date** rather than an index on **state** to perform the query.

- Place small tables or tables with restrictive filters early in the query plan. For example, consider the following query:

  ```
  SELECT * FROM customer, orders
     WHERE customer.customer_num = orders.customer_num
        AND
     customer.state = "NEVADA";
  ```

  In this example, if you read the **customer** table first, you can rule out most of the rows by applying the filter that chooses all rows in which `state = "NEVADA"`.

  By ruling out rows in the **customer** table, the database server does not read as many rows in the **orders** table (which might be significantly larger than the **customer** table).

- Choose a hash join when neither column in the join filter has an index.

  In the previous example, if **customer.customer_num** and **orders.customer_num** are not indexed, a hash join is probably the best join plan.

- Choose nested-loop joins if:
  - The number of rows retrieved from the outer table after the database server applies any table filters is small, and the inner table has an index that can be used to perform the join.
  - The index on the outermost table can be used to return rows in the order of the ORDER BY clause, eliminating the need for a sort.

Guidelines for Using Directives

Consider the following guidelines when you use directives:

- Examine the effectiveness of a particular directive frequently. Imagine a query in a production program with several directives that force an optimal query plan. Some days later, users add, update, or delete a large number of rows, which changes the nature of the data so much that the once optimal query plan is no longer effective. This example illustrates how you must use directives with care.
- Use negative directives (AVOID_NL, AVOID_FULL, and so on) whenever possible. When you exclude a behavior that degrades performance, you rely on the optimizer to use the next-best choice rather than attempt to force a path that might not be optimal.

Preparation for Using Directives

In most cases, the optimizer chooses the fastest query plan. To assist the optimizer, make sure that you perform the following tasks:

- Run UPDATE STATISTICS.

  Without accurate statistics, the optimizer cannot choose the appropriate query plan. Run UPDATE STATISTICS any time that the data in the tables changes significantly (many new rows are added, updated, or deleted). For more information, refer to "Updating Number of Rows" on page 13-14.

- Create distributions.

  One of the first things that you should try when you attempt to improve a slow query is to create distributions on columns involved in a query. Distributions provide the most accurate information to the optimizer about the nature of the data in the table. Run UPDATE STATISTICS HIGH on columns involved in the query filters to see if performance improves. For more information, refer to "Creating Data Distributions" on page 13-15.

In some cases, the query optimizer does not choose the best query plan because of the complexity of the query or because (even with distributions) it does not have enough information about the nature of the data. In these cases, you can attempt to improve performance for a particular query by using directives.

Types of Directives

Include directives in the SQL statement as a comment that occurs immediately after the SELECT, UPDATE, or DELETE keyword. The first character in a directive is always a plus (+) sign. In the following query, the ORDERED directive specifies that the tables should be joined in the same order as they are listed in the FROM clause. The AVOID_FULL directive specifies that the optimizer should discard any plans that include a full table scan on the listed table (**employee**).

```
SELECT --+ORDERED, AVOID_FULL(e) * FROM employee e, department d
> 50000;
```

For a complete syntax description for directives, refer to the *Informix Guide to SQL: Syntax*.

To influence the choice of a query plan that the optimizer makes, you can alter the following aspects of a query:

- Access plan
- Join order
- Join plan
- Optimization goal

You can also use EXPLAIN directives instead of the SET EXPLAIN statement to show the query plan. The following sections describe these aspects in detail.

Access-Plan Directives

The access plan is the method that the database server uses to access a table. The database server can either read the table sequentially (full table scan) or use any one of the indexes on the table. The following directives influence the access plan:

- INDEX

  Use the index specified to access the table. If the directive lists more than one index, the optimizer chooses the index that yields the least cost.

- AVOID_INDEX

  Do not use any of the indexes listed. You can use this directive with the AVOID_FULL directive.

- FULL

  Perform a full table scan.

- AVOID_FULL

  Do not perform a full table scan on the listed table. You can use this directive with the AVOID_INDEX directive.

In some cases, forcing an access method can change the join method that the optimizer chooses. For example, if you exclude the use of an index with the AVOID_INDEX directive, the optimizer might choose a hash join instead of a nested-loop join.

Join-Order Directives

The join-order directive ORDERED forces the optimizer to join tables in the order that the SELECT statement lists them.

Effect of Join Order on Join Plan

By specifying the join order, you might affect more than just how tables are joined. For example, consider the following query:

```
SELECT --+ORDERED, AVOID_FULL(e)
* FROM employee e, department d
WHERE e.dept_no = d.dept_no AND e.salary > 5000
```

In this example, the optimizer chooses to join the tables with a hash join. However, if you arrange the order so that the second table is **employee** (and must be accessed by an index), the hash join is not feasible.

```
SELECT --+ORDERED, AVOID_FULL(e)
* FROM department d, employee e
WHERE e.dept_no = d.dept_no AND e.salary > 5000;
```

The optimizer chooses a nested-loop join in this case.

Join Order When You Use Views

Two cases can affect join order when you use views:

- The ORDERED directive is inside the view.

  The ORDERED directive inside a view affects the join order of only the tables inside the view. The tables in the view must be joined contiguously. Consider the following view and query:

  ```
  CREATE VIEW emp_job_view as
     SELECT {+ORDERED}
     emp.job_num, job.job_name
     FROM emp, job
     WHERE emp.job_num = job.job_num;

  SELECT * from dept, emp_job_view,  project
     WHERE dept.dept_no = project.dept_num
     AND emp_job_view.job_num = project.job_num;
  ```

  The ORDERED directive specifies that the **emp** table come before the job table. The directive does not affect the order of the **dept** and **project** table. Therefore, all possible join orders are as follows:

  - **emp**, **job**, **dept**, **project**
  - **emp**, **job**, **project**, **dept**
  - **project**, **emp**, **job**, **dept**
  - **dept**, **emp**, **job**, **project**
  - **dept**, **project**, **emp**, **job**
  - **project**, **dept**, **emp**, **job**

- The ORDERED directive is in a query that contains a view.

  If an ORDERED directive appears in a query that contains a view, the join order of the tables in the query are the same as they are listed in the SELECT statement. The tables within the view are joined as they are listed within the view.

  In the following query, the join order is **dept**, **project**, **emp**, **job**:

```
CREATE VIEW emp_job_view AS
   SELECT
   emp.job_num, job.job_name
   FROM emp, job
   WHERE emp.job_num = job.job_num;
SELECT {+ORDERED}
   * FROM dept, project, emp_job_view
   WHERE dept.dept_no = project.dept_num
   AND emp_job_view.job_num = project.job_num;
```

  An exception to this rule is when the view cannot be folded into the query, as in the following example:

```
CREATE VIEW emp_job_view2 AS
   SELECT DISTINCT
   emp.job_num, job.job_name
   FROM emp,job
   WHERE emp.job_num = job.job_num;
```

  In this example, the database server executes the query and puts the result in a temporary table. The order of tables in this query is **dept**, **project**, ***temp_table***.

Join-Plan Directives

The join-plan directives influence how the database server joins two tables in a query.

The following directives influence the join plan between two tables:

- USE_NL

  Use the listed tables as the inner table in a nested-loop join.

- USE_HASH

  Access the listed tables with a hash join. You can also choose whether the table is used to create the hash table or to probe the hash table.

- AVOID_NL

  Do not use the listed tables as the inner table in a nested-loop join. A table listed with this directive can still participate in a nested-loop join as an outer table.

- AVOID_HASH

  Do not access the listed tables with a hash join. Optionally, you can allow a hash join but restrict the table from being the one that is probed or the table from which the hash table is built.

Optimization-Goal Directives

In some queries, you might want to find only the first few rows in the result of a query (for example, an ESQL/C program opens a cursor for the query and performs a FETCH to find only the first row). Or you might know that all rows must be accessed and returned from the query. You can use the optimization-goal directives to optimize the query for either one of these cases:

- FIRST_ROWS

  Choose a plan that optimizes the process of finding only the first row that satisfies the query.

- ALL_ROWS

  Choose a plan that the optimizes the process of finding all rows (the default behavior) that satisfy the query.

If you use the FIRST_ROWS directive, the optimizer might abandon a query plan that contains activities that are time-consuming upfront. For example, a hash join might take too much time to create the hash table. If only a few rows must be returned, the optimizer might choose a nested-loop join instead.

In the following example, assume that the database has an index on **employee.dept_no** but not on **department.dept_no**. Without directives, the optimizer chooses a hash join.

```
SELECT *
FROM employee, department
WHERE employee.dept_no = department.dept_no
```

However, with the FIRST_ROWS directive, the optimizer chooses a nested-loop join because of the high initial overhead required to create the hash table.

```
SELECT {+first_rows} *
FROM employee, department
WHERE employee.dept_no = department.dept_no
```

Example with Directives

The following example shows how directives can alter the query plan.

Suppose you have the following query:

```
SELECT * FROM emp,job,dept
WHERE emp.location = 10
   AND emp.jobno = job.jobno
   AND emp.deptno = dept.deptno
   AND dept.location = "DENVER";
```

Assume that the following indexes exist:

```
ix1: emp(empno,jobno,deptno,location)
ix2: job(jobno)
ix3: dept(location)
```

You run the query with SET EXPLAIN ON to display the query path that the optimizer uses.

```
QUERY:
------
SELECT * FROM emp,job,dept
WHERE emp.location = "DENVER"
   AND emp.jobno = job.jobno
   AND emp.deptno = dept.deptno
   AND dept.location = "DENVER"

Estimated Cost: 5
Estimated # of Rows Returned: 1

1) informix.emp: INDEX PATH

    Filters: informix.emp.location = 'DENVER'

    (1) Index Keys: empno jobno deptno location   (Key-Only)

2) informix.dept: INDEX PATH

    Filters: informix.dept.deptno = informix.emp.deptno

    (1) Index Keys: location
        Lower Index Filter: informix.dept.location = 'DENVER'
NESTED LOOP JOIN

3) informix.job: INDEX PATH

    (1) Index Keys: jobno   (Key-Only)
        Lower Index Filter: informix.job.jobno = informix.emp.jobno
NESTED LOOP JOIN
```

The diagram in Figure 11-1 shows a possible query plan for this query.

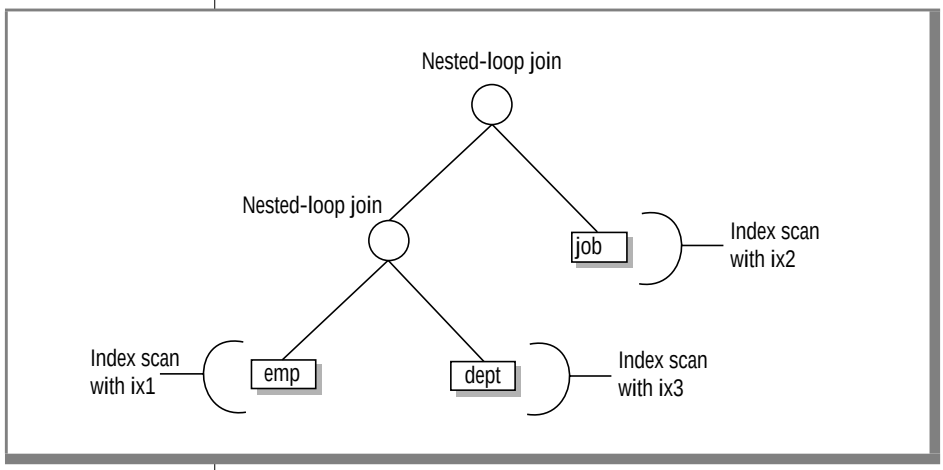


Figure 11-1
Possible Query Plan Without Directives

Perhaps you are concerned that using a nested-loop join might not be the fastest method to execute this query. You also think that the join order is not optimal. You can force the optimizer to choose a hash join and order the tables in the query plan according to their order in the query, so the optimizer uses the query plan that Figure 11-2 shows.

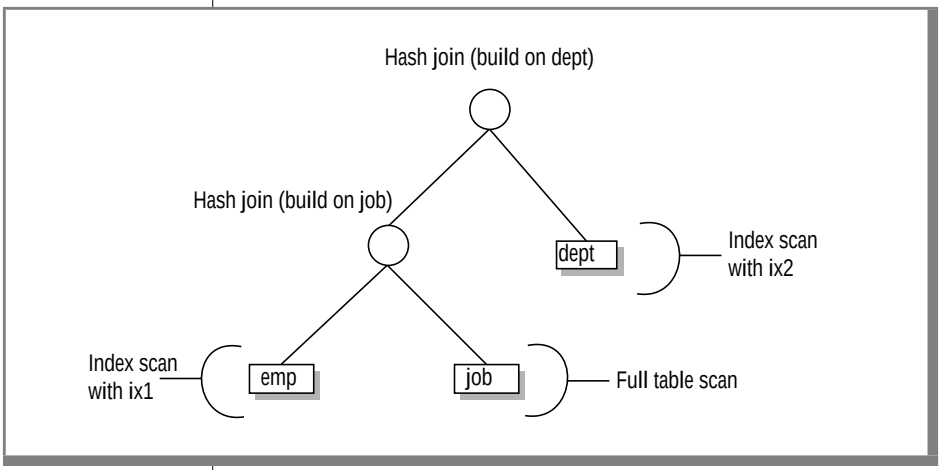


Figure 11-2
Possible Query Plan with Directives

To force the optimizer to choose the query plan that uses hash joins and the order of tables shown in the query, use the directives that the following SET EXPLAIN output shows:

```
QUERY:
------
SELECT {+ORDERED,
   INDEX(emp ix1),
   FULL(job),
   USE_HASH(job /BUILD),
   USE_HASH(dept /BUILD),
   INDEX(dept ix3)}
   * FROM emp,job,dept
   WHERE emp.location = 1
   AND emp.jobno = job.jobno
   AND emp.deptno = dept.deptno
   AND dept.location = "DENVER"

DIRECTIVES FOLLOWED:
ORDERED
INDEX ( emp ix1 )
FULL ( job )
USE_HASH ( job/BUILD )
USE_HASH ( dept/BUILD )
INDEX ( dept ix3 )
DIRECTIVES NOT FOLLOWED:

Estimated Cost: 7
Estimated # of Rows Returned: 1

1) informix.emp: INDEX PATH

    Filters: informix.emp.location = 'DENVER'

    (1) Index Keys: empno jobno deptno location   (Key-Only)

2) informix.job: SEQUENTIAL SCAN


DYNAMIC HASH JOIN
    Dynamic Hash Filters: informix.emp.jobno = informix.job.jobno

3) informix.dept: INDEX PATH

    (1) Index Keys: location
        Lower Index Filter: informix.dept.location = 'DENVER'

DYNAMIC HASH JOIN
    Dynamic Hash Filters: informix.emp.deptno = informix.dept.deptno
```

EXPLAIN Directives

You can use the EXPLAIN directives to display the query plan in the following ways:

- EXPLAIN

  Display the query plan that the optimizer chooses.

- EXPLAIN AVOID_EXECUTE

  Display the query plan that the optimizer chooses, but do not execute the query.

When you want to display the query plan for one SQL statement only, use these EXPLAIN directives instead of the SET EXPLAIN ON or SET EXPLAIN ON AVOID_EXECUTE statements.

When you use AVOID_EXECUTE (either the directive or in the SET EXPLAIN statement), the query does not execute but displays the following message:

```
No rows returned.
```

Figure 11-3 shows sample output for a query that uses the EXPLAIN AVOID_EXECUTE directive.

```
QUERY:
------
select --+ explain avoid_execute
  l.customer_num, l.lname, l.company,
  l.phone, r.call_dtime, r.call_descr
from customer l, cust_calls r
where l.customer_num = r.customer_num

DIRECTIVES FOLLOWED:
EXPLAIN
AVOID_EXECUTE
DIRECTIVES NOT FOLLOWED:

Estimated Cost: 7
Estimated # of Rows Returned: 7

  1) informix.r: SEQUENTIAL SCAN

  2) informix.l: INDEX PATH

    (1) Index Keys: customer_num   (Serial, fragments: ALL)
        Lower Index Filter: informix.l.customer_num = informix.r.customer_num
NESTED LOOP JOIN
```

Figure 11-3
Result of EXPLAIN AVOID_EXECUTE Directives

The following table describes the pertinent output lines in Figure 11-3 that describe the chosen query plan.

Output Line in Figure 11-3	Chosen Query Plan Description

`DIRECTIVES FOLLOWED:`
`EXPLAIN`
`AVOID_EXECUTE`
	Use the directives EXPLAIN AND AVOID_EXECUTE to display the query plan and do not execute the query.

`Estimated # of Rows Returned: 7`	Estimate that this query returns seven rows.
`Estimated Cost: 7`	This estimated cost of 7 is a value that the optimizer uses to compare different query plans and select the one with the lowest cost.
`1) informix.r: SEQUENTIAL SCAN`	Use the **cust_calls r** table as the outer table and scan it to obtain each row.
`2) informix.l: INDEX PATH`	For each row in the outer table, use an index to obtain the matching row(s) in the inner table **customer l**.
`(1) Index Keys: customer_num` `(Serial, fragments: ALL)`	Use the index on the **customer_num** column, scan it serially, and scan all fragments (the **customer l** table consists of only one fragment).

`Lower Index Filter:`
`informix.l.customer_num =`
`informix.r.customer_num`
	Start the index scan at the **customer_num** value from the outer table.

Configuration Parameters and Environment Variables for Directives

You can use the DIRECTIVES configuration parameter to turn on or off all directives that the database server encounters. If DIRECTIVES is `1` (the default), the optimizer follows all directives. If DIRECTIVES is `0`, the optimizer ignores all directives.

You can override the setting of DIRECTIVES with the **IFX_DIRECTIVES** environment variable. If **IFX_DIRECTIVES** is set to `1` or `ON`, the optimizer follows directives for any SQL the client session executes. If **IFX_DIRECTIVES** is `0` or `OFF`, the optimizer ignores directives for any SQL in the client session.

Any directives in an SQL statement take precedence over the join plan that the OPTCOMPIND configuration parameter forces. For example, if a query includes the USE_HASH directive and OPTCOMPIND is set to `0` (nested-loop joins preferred over hash joins), the optimizer uses a hash join.

Directives and SPL Routines

Directives operate differently for a query in an SPL routine because a SELECT statement in an SPL routine is not necessarily optimized immediately before the database server executes it. The optimizer creates a query plan for a SELECT statement in an SPL routine when the database server creates an SPL routine or during the execution of some versions of the UPDATE STATISTICS statement.

The optimizer reads and applies directives at the time that it creates the query plan. Because it stores the query plan in a system catalog table, the SELECT statement is not reoptimized when it is executed. Therefore, settings of **IFX_DIRECTIVES** and DIRECTIVES affect SELECT statements inside an SPL routine when they are set at any of the following times:

- Before the CREATE PROCEDURE statement
- Before the UPDATE STATISTICS statements that cause SQL in SPL to be optimized
- During certain circumstances when SELECT statements have variables supplied at runtime

Chapter

12

Parallel Database Query

In This Chapter 12-3

Structure of a Parallel Database Query 12-3

Database Server Operations That Use PDQ 12-5
- Parallel Delete 12-5
- Parallel Inserts 12-5
  - Explicit Inserts with SELECT...INTO TEMP 12-5
  - Implicit Inserts with INSERT INTO...SELECT 12-6
- Parallel Index Builds 12-7
- Parallel User-Defined Routines 12-8
- SQL Operations That Do Not Use PDQ 12-8
- Update Statistics 12-9
- SPL Routines and Triggers 12-9
- Correlated and Uncorrelated Subqueries 12-9
- OUTER Index Joins 12-10
- Remote Tables 12-10

Memory Grant Manager 12-10

Allocating Resources for Parallel Database Queries 12-12
- Limiting the Priority of DSS Queries 12-13
  - Limiting the Value of PDQ Priority 12-14
  - Maximizing OLTP Throughput 12-15
  - Conserving Resources 12-15
  - Allowing Maximum Use of Parallelism 12-15
  - Determining the Level of Parallelism 12-16
  - Limits on Parallelism Associated with PDQ Priority 12-16
  - Using SPL Routines 12-16

Adjusting the Amount of Memory 12-17
Limiting the Number of Concurrent Scans 12-18
Limiting the Maximum Number of Queries 12-19

Managing Applications 12-19
Using SET EXPLAIN 12-19
Using OPTCOMPIND 12-20
Using SET PDQPRIORITY 12-20
User Control of Resources 12-21
DBA Control of Resources 12-21
Controlling Resources Allocated to PDQ 12-21
Controlling Resources Allocated to Decision-Support Queries 12-22

Monitoring PDQ Resources 12-23
Using the onstat Utility 12-23
Monitoring MGM Resources 12-23
Monitoring PDQ Threads 12-27
Monitoring Resources Allocated for a Session 12-29
Using SET EXPLAIN 12-29

In This Chapter

Parallel database query (PDQ) is an Informix database server feature that can improve performance dramatically when the database server processes queries that decision-support applications initiate. PDQ allows the database server to distribute the work for one aspect of a query among several processors. For example, if a query requires an aggregation, the database server can distribute the work for the aggregation among several processors. PDQ also includes tools for resource management.

Another database server feature, *table fragmentation*, allows you to store the parts of a table on different disks. PDQ delivers maximum performance benefits when the data that you query is in fragmented tables. For information on how to use fragmentation for maximum performance, refer to "Planning a Fragmentation Strategy" on page 9-3.

This chapter discusses the parameters and strategies that you use to manage resources for PDQ.

Structure of a Parallel Database Query

Each decision-support query has a primary thread. The database server can start additional threads to perform tasks for the query (for example, scans and sorts). Depending on the number of tables or fragments that a query must search and the resources that are available for a decision-support query, the database server assigns different components of a query to different threads. The database server initiates these PDQ threads, which are listed as *secondary threads* in the SET EXPLAIN output.

Secondary threads are further classified as either *producers* or *consumers*, depending on their function. A producer thread supplies data to another thread. For example, a scan thread might read data from shared memory that corresponds to a given table and pass it along to a join thread. In this case, the scan thread is considered a producer, and the join thread is considered a consumer. The join thread, in turn, might pass data along to a sort thread. When doing so, the join thread is considered a producer, and the sort thread is considered a consumer.

Several producers can supply data to a single consumer. When this situation occurs, the database server sets up an internal mechanism, called an *exchange*, that synchronizes the transfer of data from those producers to the consumer. For instance, if a fragmented table is to be sorted, the optimizer typically calls for a separate scan thread for each fragment. Because of different I/O characteristics, the scan threads can be expected to complete at different times. An exchange is used to funnel the data produced by the various scan threads into one or more sort threads with a minimum of buffering. Depending on the complexity of the query, the optimizer might call for a multilayered hierarchy of producers, exchanges, and consumers. Generally speaking, consumer threads work in parallel with producer threads so that the amount of intermediate buffering that the exchanges perform remains negligible.

The database server creates these threads and exchanges automatically and transparently. They terminate automatically as they complete processing for a given query. The database server creates new threads and exchanges as needed for subsequent queries.

Database Server Operations That Use PDQ

This section describes the types of SQL operations that the database server processes in parallel and the situations that limit the degree of parallelism that the database server can use. In the following discussions, a *query* is any SELECT statement.

Parallel Delete

The database server takes the following two steps to process DELETE, INSERT, and UPDATE statements:

1. Fetches the qualifying rows
2. Applies the action of deleting, inserting, or updating

The database server performs the first step of a DELETE statement in parallel, with one exception; the database server does not process the first part of a DELETE statement in parallel if the targeted table has a referential constraint that can cascade to a child table.

Parallel Inserts

The database server performs the following types of inserts in parallel:

- SELECT...INTO TEMP inserts using explicit temporary tables.
- INSERT INTO...SELECT inserts using implicit temporary tables.

For information on implicit and explicit temporary tables, refer to the chapter on where data is stored in the *Administrator's Guide*.

Explicit Inserts with SELECT…INTO TEMP

The database server can insert rows in parallel into explicit temporary tables that you specify in SQL statements of the form SELECT....INTO TEMP. For example, the database server can perform the inserts in parallel into the temporary table, **temp_table**, as the following example shows:

```
SELECT * FROM table1 INTO TEMP temp_table
```

To perform parallel inserts into a temporary table

1. Set PDQ priority > 0.

   You must meet this requirement for any query that you want the database server to perform in parallel.

2. Set DBSPACETEMP to a list of two or more dbspaces.

   This step is required because of the way that the database server performs the insert. To perform the insert in parallel, the database server first creates a fragmented temporary table. So that the database server knows where to store the fragments of the temporary table, you must specify a list of two or more dbspaces in the DBSPACETEMP configuration parameter or the **DBSPACETEMP** environment variable. In addition, you must set DBSPACETEMP to indicate storage space for the fragments before you execute the SELECT...INTO statement.

The database server performs the parallel insert by writing in parallel to each of the fragments in a round-robin fashion. Performance improves as you increase the number of fragments.

Implicit Inserts with INSERT INTO…SELECT

The database server can also insert rows in parallel into implicit tables that it creates when it processes SQL statements of the form INSERT INTO...SELECT. For example, the database server processes the following INSERT statement in parallel:

```
INSERT INTO target_table SELECT * FROM source_table
```

The target table can be either a permanent table or a temporary table.

The database server processes this type of INSERT statement in parallel only when the target tables meet the following criteria:

- The value of PDQ priority is greater than 0.
- The target table is fragmented into two or more dbspaces.
- The target table has no enabled referential constraints or triggers.

- The target table is not a remote table.
- In a database with logging, the target table does not contain filtering constraints.
- The target table does not contain columns of TEXT or BYTE data type.

The database server does not process parallel inserts that reference an SPL routine. For example, the database server never processes the following statement in parallel:

```
INSERT INTO table1 EXECUTE PROCEDURE ins_proc
```

Parallel Index Builds

Index builds can take advantage of PDQ and can be parallelized. The database server performs both scans and sorts in parallel for index builds. The following operations initiate index builds:

- Create an index.
- Add a unique, primary key.
- Add a referential constraint.
- Enable a referential constraint.

When PDQ is in effect, the scans for index builds are controlled by the PDQ configuration parameters described in "Allocating Resources for Parallel Database Queries" on page 12-12.

If you have a computer with multiple CPUs, the database server uses two sort threads to sort the index keys. The database server uses two sort threads during index builds without the user setting the **PSORT_NPROCS** environment variable.

Parallel User-Defined Routines

If a query contains a user-defined routine (UDR) in an expression, the database server can execute a query in parallel when you turn on PDQ. The database server can perform the following parallel operations if the UDR is written and registered appropriately:

- Parallel scans
- Parallel comparisons with the UDR

For more information on how to enable parallel execution of UDRs, refer to "Parallel UDRs" on page 13-38.

SQL Operations That Do Not Use PDQ

The database server does not process the following types of queries in parallel:

- Queries started with an isolation level of Cursor Stability

  Subsequent changes to the isolation level do not affect the parallelism of queries already prepared. This situation results from the inherent nature of parallel scans, which scan several rows simultaneously.
- Queries that use a cursor declared as FOR UPDATE or with the WITH HOLD qualifier
- An UPDATE statement that has an *update* trigger that updates in the `For Each Row` section of the trigger definition
- Data definition language (DDL) statements

  For a complete list, see the *Informix Guide to SQL: Syntax*.

Update Statistics

The SQL UPDATE STATISTICS statement, which is not processed in parallel, is affected by PDQ because it must allocate the memory used for sorting. Thus the behavior of the UPDATE STATISTICS statement is affected by the memory management associated with PDQ.

Even though the UPDATE STATISTICS statement is not processed in parallel, the database server must allocate the memory that this statement uses for sorting.

SPL Routines and Triggers

Statements that involve SPL routines are not executed in parallel. However, statements within procedures are executed in parallel.

When the database server executes an SPL routine, it does not use PDQ to process nonrelated SQL statements contained in the procedure. However, each SQL statement can be executed independently in parallel, using intraquery parallelism when possible. As a consequence, you should limit the use of procedure calls from within data manipulation language (DML) statements if you want to exploit the parallel-processing abilities of the database server. For a complete list of DML statements, see the *Informix Guide to SQL: Syntax*.

The database server uses intraquery parallelism to process the statements in the body of an SQL trigger in the same way that it processes statements in SPL routines.

Correlated and Uncorrelated Subqueries

The database server does not use PDQ to process correlated subqueries. Only one thread at a time can execute a correlated subquery. While one thread executes a correlated subquery, other threads that request to execute the subquery are blocked until the first one completes.

For uncorrelated subqueries, only the first thread that makes the request actually executes the subquery. Other threads simply use the results of the subquery and can do so in parallel.

As a consequence, Informix strongly recommends that, whenever possible, you use joins rather than subqueries to build queries so that the queries can take advantage of PDQ.

OUTER Index Joins

The database server reduces the PDQ priority of queries that contain OUTER index joins to LOW (if the priority is set to a higher value) for the duration of the query. If a subquery or a view contains OUTER index joins, the database server lowers the PDQ priority of only that subquery or view, not of the parent query or any other subquery.

Remote Tables

Although the database server can process the data stored in a remote table in parallel, the data is communicated serially because the database server allocates a single thread to submit and receive the data from the remote table.

The database server lowers the PDQ priority of queries that require access to a remote database to LOW. In that case, all local scans are parallel, but all local joins and remote access are nonparallel.

Memory Grant Manager

The Memory Grant Manager (MGM) is a database server component that coordinates the use of memory, CPU virtual processors (VPs), disk I/O, and scan threads among decision-support queries. The MGM uses the DS_MAX_QUERIES, DS_TOTAL_MEMORY, DS_MAX_SCANS, and MAX_PDQPRIORITY configuration parameters to determine the quantity of these PDQ resources that can be granted to a decision-support query. For more information about these configuration parameters, refer to Chapter 4, “Effect of Configuration on Memory Utilization.”

The MGM dynamically allocates the following resources for decision-support queries:

- The number of scan threads started for each decision-support query
- The number of threads that can be started for each query
- The amount of memory in the virtual portion of database server shared memory that the query can reserve

When your database server system has heavy OLTP use, and you find performance is degrading, you can use the MGM facilities to limit the resources committed to decision-support queries. During off-peak hours, you can designate a larger proportion of the resources to parallel processing, which achieves higher throughput for decision-support queries.

The MGM grants memory to a query for such activities as sorts, hash joins, and processing of GROUP BY clauses. The amount of memory that decision-support queries use cannot exceed DS_TOTAL_MEMORY.

The MGM grants memory to queries in *quantum* increments. To calculate a quantum, use the following formula:

```
memory quantum = DS_TOTAL_MEMORY / DS_MAX_QUERIES
```

For example, if DS_TOTAL_MEMORY is 12 megabytes and DS_MAX_QUERIES is 4, the quantum is 3 megabytes (12 / 4). Thus, with these values in effect, a quantum of memory equals 3 megabytes. In general, memory is allocated more efficiently when quanta are smaller. You can often improve performance of concurrent queries by increasing DS_MAX_QUERIES to reduce the size of a quantum of memory.

To monitor resources that the MGM allocates, run the **onstat -g mgm** command. This command displays only the amount of memory that is currently used; it does not display the amount of memory that has been granted. For more information about this command, refer to "Monitoring MGM Resources" on page 12-23.

The MGM also grants a maximum number of scan threads per query based on the values of the DS_MAX_SCANS and the DS_MAX_QUERIES parameters.

The following formula yields the maximum number of scan threads per query:

```
scan_threads = min (nfrags, DS_MAX_SCANS * (pdqpriority / 100)
   * (MAX_PDQPRIORITY / 100))
```

nfrags is the number of fragments in the table with the largest number of fragments.

pdqpriority is the value for PDQ priority that is set by either the **PDQPRIORITY** environment variable or the SQL statement SET PDQPRIORITY.

For more information about any of these database server configuration parameters, refer to Chapter 4, "Effect of Configuration on Memory Utilization."

The **PDQPRIORITY** environment variable and the SQL statement SET PDQPRIORITY request a percentage of PDQ resources for a query. You can use the MAX_PDQPRIORITY configuration parameter to limit the percentage of the requested resources that a query can obtain and to limit the impact of decision-support queries on OLTP processing. For more information, refer to "Limiting the Priority of DSS Queries" on page 12-13.

Allocating Resources for Parallel Database Queries

When the database server uses PDQ to perform a query in parallel, it puts a heavy load on the operating system. In particular, PDQ exploits the following resources:

- Memory
- CPU VPs
- Disk I/O (to fragmented tables and temporary table space)
- Scan threads

When you configure the database server, consider how the use of PDQ affects users of OLTP, decision-support applications, and other applications.

You can control how the database server uses resources in the following ways:

- Limit the priority of parallel database queries.
- Adjust the amount of memory.
- Limit the number of scan threads.
- Limit the number of concurrent queries.

Limiting the Priority of DSS Queries

The default value for the PDQ priority of individual applications is `0`, which means that PDQ processing is not used. The database server uses this value unless one of the following actions overrides it:

- The user sets the **PDQPRIORITY** environment variable.
- The application uses the SET PDQPRIORITY statement.

The **PDQPRIORITY** environment variable and the MAX_PDQPRIORITY configuration parameter work together to control the amount of resources to allocate for parallel processing. Setting these configuration parameters correctly is critical for the effective operation of PDQ.

The MAX_PDQPRIORITY configuration parameter allows the database server administrator to limit the parallel processing resources that DSS queries consume. Thus, the **PDQPRIORITY** environment variable sets a *reasonable* or *recommended* priority value, and MAX_PDQPRIORITY limits the resources that an application can claim.

The MAX_PDQPRIORITY configuration parameter specifies the maximum percentage of the requested resources that a query can obtain. For instance, if **PDQPRIORITY** is `80` and MAX_PDQPRIORITY is `50`, each active query receives an amount of memory equal to 40 percent of DS_TOTAL_MEMORY, rounded down to the nearest quantum. In this example, MAX_PDQPRIORITY effectively limits the number of concurrent decision-support queries to two. Subsequent queries must wait for earlier queries to finish before they acquire the resources that they need to run.

An application or user can use the DEFAULT tag of the SET PDQPRIORITY statement to use the value for PDQ priority if the value has been set by the **PDQPRIORITY** environment variable. DEFAULT is the symbolic equivalent of a `-1` value for PDQ priority.

You can use the **onmode** command-line utility to change the values of the following configuration parameters temporarily:

- Use **onmode -M** to change the value of DS_TOTAL_MEMORY.
- Use **onmode -Q** to change the value of DS_MAX_QUERIES.
- Use **onmode -D** to change the value of MAX_PDQPRIORITY.
- Use **onmode -S** to change the value of DS_MAX_SCANS.

These changes remain in effect only as long as the database server remains up and running. When the database server is initialized, it uses the values listed in the ONCONFIG file.

For more information about the preceding parameters, refer to Chapter 4, "Effect of Configuration on Memory Utilization." For more information about **onmode**, refer to your *Administrator's Reference*.

If you must change the values of the decision-support parameters on a regular basis (for example, to set MAX_PDQPRIORITY to `100` each night for processing reports), you can use a scheduled operating-system job to set the values. For information about creating scheduled jobs, refer to your operating-system manuals.

To obtain the best performance from the database server, choose values for the **PDQPRIORITY** environment variable and MAX_PDQPRIORITY parameter, observe the resulting behavior, and then adjust the values for these parameters. No well-defined rules exist for choosing these environment variable and parameter values. The following sections discuss strategies for setting **PDQPRIORITY** and MAX_PDQPRIORITY for specific needs.

Limiting the Value of PDQ Priority

The MAX_PDQPRIORITY configuration parameter limits the PDQ priority that the database server grants when users either set the **PDQPRIORITY** environment variable or issue the SET PDQPRIORITY statement before they issue a query. When an application or an end user attempts to set a PDQ priority, the priority that is granted is multiplied by the value that MAX_PDQPRIORITY specifies.

Set the value of MAX_PDQPRIORITY lower when you want to allocate more resources to OLTP processing. Set the value higher when you want to allocate more resources to decision-support processing. The possible range of values is 0 to 100. This range represents the percent of resources that you can allocate to decision-support processing.

Maximizing OLTP Throughput

At times, you might want to allocate resources to maximize the throughput for individual OLTP queries rather than for decision-support queries. In this case, set MAX_ PDQPRIORITY to `0`, which limits the value of PDQ priority to `OFF`. A PDQ priority value of `OFF` does not prevent decision-support queries from running. Instead, it causes the queries to run without parallelization. In this configuration, response times for decision-support queries might be slow.

Conserving Resources

If applications make little use of queries that require parallel sorts and parallel joins, consider using the `LOW` setting for PDQ priority.

If the database server is operating in a multiuser environment, you might set MAX_PDQPRIORITY to `1` to increase interquery performance at the cost of some intraquery parallelism. A trade-off exists between these two different types of parallelism because they compete for the same resources. As a compromise, you might set MAX_PDQPRIORITY to some intermediate value (perhaps `20` or `30`) and set **PDQPRIORITY** to `LOW`. The environment variable sets the default behavior to `LOW`, but the MAX_PDQPRIORITY configuration parameter allows individual applications to request more resources with the SET PDQPRIORITY statement.

Allowing Maximum Use of Parallelism

Set **PDQPRIORITY** and MAX_PDQPRIORITY to `100` if you want the database server to assign as many resources as possible to parallel processing. This setting is appropriate for times when parallel processing does not interfere with OLTP processing.

Determining the Level of Parallelism

You can use different numeric settings for **PDQPRIORITY** to experiment with the effects of parallelism on a single application. For information on how to monitor parallel execution, refer to "Monitoring PDQ Resources" on page 12-23.

Limits on Parallelism Associated with PDQ Priority

The database server reduces the PDQ priority of queries that contain outer joins to `LOW` (if set to a higher value) for the duration of the query. If a subquery or a view contains outer joins, the database server lowers the PDQ priority only of that subquery or view, not of the parent query or of any other subquery.

The database server lowers the PDQ priority of queries that require access to a remote database (same or different database server instance) to `LOW` if you set it to a higher value. In that case, all local scans are parallel, but all local joins and remote accesses are nonparallel.

Using SPL Routines

The database server freezes the PDQ priority that is used to optimize SQL statements within SPL routines at the time of procedure creation or the last manual recompilation with the UPDATE STATISTICS statement. To change the client value of **PDQPRIORITY**, embed the SET PDQPRIORITY statement within the body of your SPL routine.

The PDQ priority value that the database server uses to optimize or reoptimize an SQL statement is the value that was set by a SET PDQPRIORITY statement, which must have been executed within the same procedure. If no such statement has been executed, the value that was in effect when the procedure was last compiled or created is used.

The PDQ priority value currently in effect outside a procedure is ignored within a procedure when it is executing.

Informix suggests that you turn PDQ priority off when you enter a procedure and then turn it on again for specific statements. You can avoid tying up large amounts of memory for the procedure, and you can make sure that the crucial parts of the procedure use the appropriate PDQ priority, as the following example illustrates:

```
CREATE PROCEDURE my_proc (a INT, b INT, c INT)
   Returning INT, INT, INT;
SET PDQPRIORITY 0;
...
SET PDQPRIORITY 85;
SELECT ... (big complicated SELECT statement)
SET PDQPRIORITY 0;
...
;
```

Adjusting the Amount of Memory

Use the following formula as a starting point for estimating the amount of shared memory to allocate to decision-support queries:

```
DS_TOTAL_MEMORY = p_mem - os_mem - rsdnt_mem -
                  (128 kilobytes * users) - other_mem
```

p_mem represents the total physical memory that is available on the host computer.

os_mem represents the size of the operating system, including the buffer cache.

resdnt_mem represents the size of Informix resident shared memory.

users is the number of expected users (connections) specified in the NETTYPE configuration parameter.

other_mem is the size of memory used for other (non-Informix) applications.

The value for DS_TOTAL_MEMORY that is derived from this formula serves only as a starting point. To arrive at a value that makes sense for your configuration, you must monitor paging and swapping. (Use the tools provided with your operating system to monitor paging and swapping.) When paging increases, decrease the value of DS_TOTAL_MEMORY so that processing the OLTP workload can proceed.

The amount of memory that is granted to a single parallel database query depends on many system factors, but in general, the amount of memory granted to a single parallel database query is proportional to the following formula:

```
memory_grant_basis = (DS_TOTAL_MEMORY/DS_MAX_QUERIES) *
                     (PDQPRIORITY / 100) *
                     (MAX_PDQPRIORITY / 100)
```

Limiting the Number of Concurrent Scans

The database server apportions some number of scans to a query according to its PDQ priority (among other factors). DS_MAX_SCANS and MAX_PDQPRIORITY allow you to limit the resources that users can assign to a query, according to the following formula:

```
scan_threads = min (nfrags, (DS_MAX_SCANS * (pdqpriority / 100)
   * (MAX_PDQPRIORITY / 100) )
```

nfrags is the number of fragments in the table with the largest number of fragments.

pdqpriority is the PDQ priority value set by either the **PDQPRIORITY** environment variable or the SET PDQPRIORITY statement.

For example, suppose a large table contains 100 fragments. With no limit on the number of concurrent scans allowed, the database server would concurrently execute 100 scan threads to read this table. In addition, as many users as wanted to could initiate this query.

As the database server administrator, you set DS_MAX_SCANS to a value lower than the number of fragments in this table to prevent the database server from being flooded with scan threads by multiple decision-support queries. You can set DS_MAX_SCANS to 20 to ensure that the database server concurrently executes a maximum of 20 scan threads for parallel scans. Furthermore, if multiple users initiate parallel database queries, each query receives only a percentage of the 20 scan threads, according to the PDQ priority assigned to the query and the value for MAX_PDQPRIORITY that the database server administrator sets.

Limiting the Maximum Number of Queries

The DS_MAX_QUERIES configuration parameter limits the number of concurrent decision-support queries that can run. To estimate the number of decision-support queries that the database server can run concurrently, count each query that runs with PDQ priority set to 1 or greater as one full query.

The database server allocates less memory to queries that run with a lower priority, so you can assign lower-priority queries a PDQ priority value that is between 1 and 30, depending on the resource impact of the query. The total number of queries with PDQ priority values greater than 0 cannot exceed DS_MAX_QUERIES.

Managing Applications

The database server administrator, the writer of an application, and the users all have a certain amount of control over the amount of resources that the database server allocates to processing a query. The database server administrator exerts control through the use of configuration parameters. The application developer or the user can exert control through an environment variable or SQL statement.

Using SET EXPLAIN

The output of the SET EXPLAIN statement shows decisions that the query optimizer makes. It shows whether parallel scans are used, the maximum number of threads required to answer the query, and the type of join used for the query. You can use SET EXPLAIN to study the query plans of an application. You can restructure a query or use OPTCOMPIND to change how the optimizer treats the query.

Using OPTCOMPIND

The **OPTCOMPIND** environment variable and the OPTCOMPIND configuration parameter indicate the preferred join plan, thus assisting the optimizer in selecting the appropriate join method for parallel database queries.

To influence the optimizer in its choice of a join plan, you can set the OPTCOMPIND configuration parameter. The value that you assign to this configuration parameter is referenced only when applications do not set the **OPTCOMPIND** environment variable.

You can set OPTCOMPIND to `0` if you want the database server to select a join plan exactly as it did in versions of the database server prior to 6.0. This option ensures compatibility with previous versions of the database server.

An application with an isolation mode of Repeatable Read can lock all records in a table when it performs a hash join. For this reason, Informix recommends that you set OPTCOMPIND to `1`.

If you want the optimizer to make the determination for you based on cost, regardless of the isolation level of applications, set OPTCOMPIND to `2`.

For more information on OPTCOMPIND and the different join plans, refer to "The Query Plan" on page 10-3.

Using SET PDQPRIORITY

The SET PDQPRIORITY statement allows you to set PDQ priority dynamically within an application. The PDQ priority value can be any integer from -1 through 100.

The PDQ priority set with the SET PDQPRIORITY statement supersedes the **PDQPRIORITY** environment variable.

The DEFAULT tag for the SET PDQPRIORITY statement allows an application to revert to the value for PDQ priority as set by the environment variable, if any. For more information about the SET PDQPRIORITY statement, refer to the *Informix Guide to SQL: Syntax*.

User Control of Resources

To indicate the PDQ priority of a query, a user sets the **PDQPRIORITY** environment variable or executes the SET PDQPRIORITY statement prior to issuing a query. In effect, the PDQ priority allows users to request a certain amount of parallel-processing resources for the query.

The resources that a user requests and the amount that the database server allocates for the query can differ. This difference occurs when the database server administrator uses the MAX_PDQPRIORITY configuration parameter to put a ceiling on user-requested resources, as the following section explains.

DBA Control of Resources

To manage the total amount of resources that the database server allocates to parallel database queries, the database server administrator sets the environment variable and configuration parameters that the following sections discuss.

Controlling Resources Allocated to PDQ

To control resources allocated to PDQ, you can set the **PDQPRIORITY** environment variable. The queries that do not set the **PDQPRIORITY** environment variable before they issue a query do not use PDQ. In addition, to place a ceiling on user-specified PDQ priority levels, you can set the MAX_PDQPRIORITY configuration parameter.

When you set the **PDQPRIORITY** environment variable and MAX_PDQPRIORITY parameter, you exert control over the resources that the database server allocates between OLTP and DSS applications. For example, if OLTP processing is particularly heavy during a certain period of the day, you might want to set MAX_PDQPRIORITY to `0`. This configuration parameter puts a ceiling on the resources requested by users who use the **PDQPRIORITY** environment variable, so PDQ is turned off until you reset MAX_PDQPRIORITY to a nonzero value.

Controlling Resources Allocated to Decision-Support Queries

To control the resources that the database server allocates to decision-support queries, set the DS_TOTAL_MEMORY, DS_MAX_SCANS, and DS_MAX_QUERIES configuration parameters. In addition to setting limits for decision-support memory and the number of decision-support queries that can run concurrently, the database server uses these parameters to determine the amount of memory to allocate to individual decision-support queries as users submit them. To do so, the database server first calculates a unit of memory called a *quantum* by dividing DS_TOTAL_MEMORY by DS_MAX_QUERIES. When a user issues a query, the database server allocates a percent of the available quanta equal to the PDQ priority of the query.

You can also limit the number of concurrent decision-support scans that the database server allows by setting the DS_MAX_SCANS configuration parameter.

Previous versions of the database server allowed you to set a PDQ priority configuration parameter in the ONCONFIG file. If your applications depend on a global setting for PDQ priority, you can use one of the following methods:

UNIX

- Define **PDQPRIORITY** as a shared environment variable in the informix.rc file. For more information on the informix.rc file, see the *Informix Guide to SQL: Reference.* ♦

WIN NT

- Set the **PDQPRIORITY** environment variable for a particular group through a logon profile. For more information on the logon profile, see your operating-system manual. ♦

Monitoring PDQ Resources

Monitor the resources (shared memory and threads) that the MGM has allocated for PDQ queries and the resources that those queries currently use.

You monitor PDQ resource use in the following ways:

- Run individual **onstat** utility commands to capture information about specific aspects of a running query.
- Execute a SET EXPLAIN statement before you run a query to write the query plan to an output file.

Using the onstat Utility

You can use various **onstat** utility commands to determine how many threads are active and the shared-memory resources that those threads use.

Monitoring MGM Resources

You can use the **onstat -g mgm** option to monitor how MGM coordinates memory use and scan threads. The **onstat** utility reads shared-memory structures and provides statistics that are accurate at the instant that the command executes. Figure 12-1 on page 12-24 shows sample output.

The **onstat -g mgm** output displays a unit of memory called a *quantum*. The *memory quantum* represents a unit of memory, as follows:

```
memory quantum = DS_TOTAL_MEMORY / DS_MAX_QUERIES
```

The following calculation shows the memory quantum for the values that Figure 12-1 on page 12-24 displays:

```
memory quantum = 4000 kilobytes / 5
               = 800 kilobytes
```

The *scan thread quantum* is always equal to 1.

```
Memory Grant Manager (MGM)
--------------------------

MAX_PDQPRIORITY:  100
DS_MAX_QUERIES:    5
DS_MAX_SCANS:      10
DS_TOTAL_MEMORY:   4000 KB

Queries:   Active     Ready   Maximum
                3         0         5

Memory:     Total      Free   Quantum
(KB)         4000      3872       800

Scans:      Total      Free   Quantum
               10         8         1

Load Control:    (Memory)      (Scans)  (Priority)  (Max Queries)   (Reinit)
                   Gate 1       Gate 2      Gate 3         Gate 4     Gate 5
(Queue Length)          0            0           0              0          0

Active Queries:
---------------
Session   Query  Priority  Thread   Memory  Scans      Gate
      7  a3d0c0      1     a8adcc    0/0       1/1       -
      7  a56eb0      1     ae6800    0/0       1/1       -
      9  a751d4      0     96b1b8   16/16      0/0       -

Ready Queries:  None

Free Resource        Average #        Minimum #
--------------    ---------------     ---------
Memory             489.2 +- 28.7          400
Scans                8.5 +- 0.5             8

Queries              Average #        Maximum #    Total #
--------------    ---------------     ---------    -------
Active               1.7 +- 0.7             3         23
Ready                0.0 +- 0.0             0          0

Resource/Lock Cycle Prevention count:  0
```

Figure 12-1
onstat -g mgm Output

The first portion of the output shows the values of the PDQ configuration parameters.

The second portion of the output describes MGM internal control information. It includes four groups of information.

The first group is **Queries.**

Column	Description
Active	Number of PDQ queries that are currently executing
Ready	Number of user queries ready to run but whose execution the database server deferred for load-control reasons
Maximum	Maximum number of queries that the database server permits to be active. Reflects current value of the DS_MAX_QUERIES configuration parameter

The next group is **Memory**.

Column	Description
Total	Kilobytes of memory available for use by PDQ queries (DS_TOTAL_MEMORY specifies this value.)
Free	Kilobytes of memory for PDQ queries not currently in use
Quantum	Kilobytes of memory in a memory quantum

The next group is **Scans**.

Column	Description
Total	The total number of scan threads as specified by the DS_MAX_SCANS configuration parameter
Free	Number of scan threads currently available for decision-support queries
Quantum	The number of scan threads in a scan-thread quantum

The last group in this portion of the output describes MGM **Load Control**.

Column	Description
Memory	Number of queries that are waiting for memory
Scans	Number of queries that are waiting for scans
Priority	Number of queries that are waiting for queries with higher PDQ priority to run
Max Queries	Number of queries that are waiting for a query slot
Reinit	Number of queries that are waiting for running queries to complete after an **onmode -M** or **-Q** command

The next portion of the output, **Active Queries**, describes the MGM active and ready queues. This portion of the output shows the number of queries waiting at each gate.

Column	Description
Session	The session ID for the session that initiated the query
Query	Address of the internal control block associated with the query
Priority	PDQ priority assigned to the query
Thread	Thread that registered the query with MGM
Memory	Memory currently granted to the query or memory reserved for the query (Unit is MGM pages, which is 8 kilobytes.)
Scans	Number of scan threads currently used by the query or number of scan threads allocated to the query
Gate	Gate number at which query is waiting

The next portion of the output, **Free Resource**, provides statistics for MGM free resources. The numbers in this portion and in the final portion reflect statistics since system initialization or the last **onmode -Q**, **-M**, or **-S** command.

Column	Description
Average	Average amount of memory and number of scans
Minimum	Minimum available memory and number of scans

The last portion of the output, **Queries**, provides statistics concerning MGM queries.

Column	Description
Average	Average active and ready queue length
Minimum	Minimum active and ready queue length
Total	Total active and ready queue length

Monitoring PDQ Threads

To obtain information on all of the threads that are running for a decision-support query, use the **onstat -u** and **onstat -g ath** options.

The **onstat -u** option lists all the threads for a session. If a session is running a decision-support query, the output lists the primary thread and any additional threads. For example, session `10` in Figure 12-2 has a total of five threads running.

```
Userthreads
address  flags   sessid   user     tty      wait     tout locks nreads
nwrites
80eb8c   ---P--D 0        informix -        0        0    0     33       19
80ef18   ---P--F 0        informix -        0        0    0     0        0
80f2a4   ---P--B 3        informix -        0        0    0     0        0
80f630   ---P--D 0        informix -        0        0    0     0        0
80fd48   ---P--- 45       chrisw   ttyp3    0        0    1     573      237
810460   ------- 10       chrisw   ttyp2    0        0    1     1        0
810b78   ---PR-- 42       chrisw   ttyp3    0        0    1     595      243
810f04   Y------ 10       chrisw   ttyp2    beacf8   0    1     1        0
811290   ---P--- 47       chrisw   ttyp3    0        0    2     585      235
81161c   ---PR-- 46       chrisw   ttyp3    0        0    1     571      239
8119a8   Y------ 10       chrisw   ttyp2    a8a944   0    1     1        0
81244c   ---P--- 43       chrisw   ttyp3    0        0    2     588      230
8127d8   ----R-- 10       chrisw   ttyp2    0        0    1     1        0
812b64   ---P--- 10       chrisw   ttyp2    0        0    1     20       0
812ef0   ---PR-- 44       chrisw   ttyp3    0        0    1     587      227
 15 active, 20 total, 17 maximum concurrent
```

Figure 12-2
onstat -u Output

The **onstat -g ath** output also lists these threads and includes a **name** column that indicates the role of the thread. Threads that a primary decision-support thread started have a name that indicates their role in the decision-support query. For example, Figure 12-3 lists four *scan* threads, started by a primary thread (**sqlexec**).

```
Threads:
tid      tcb      rstcb    prty    status                  vp-class   name
...
11       994060   0        4       sleeping(Forever)         1cpu     kaio
12       994394   80f2a4   2       sleeping(secs: 51)        1cpu     btclean
26       99b11c   80f630   4       ready                     1cpu     onmode_mon
32       a9a294   812b64   2       ready                     1cpu     sqlexec
113      b72a7c   810b78   2       ready                     1cpu     sqlexec
114      b86c8c   81244c   2       cond wait(netnorm)        1cpu     sqlexec
115      b98a7c   812ef0   2       cond wait(netnorm)        1cpu     sqlexec
116      bb4a24   80fd48   2       cond wait(netnorm)        1cpu     sqlexec
117      bc6a24   81161c   2       cond wait(netnorm)        1cpu     sqlexec
118      bd8a24   811290   2       ready                     1cpu     sqlexec
119      beae88   810f04   2       cond wait(await_MC1)      1cpu     scan_1.0
120      a8ab48   8127d8   2       ready                     1cpu     scan_2.0
121      a96850   810460   2       ready                     1cpu     scan_2.1
122      ab6f30   8119a8   2       running                   1cpu     scan_2.2
```

Figure 12-3
onstat -g ath Output

Monitoring Resources Allocated for a Session

Use the **onstat -g ses** option to monitor the resources allocated for, and used by, a session that is running a decision-support query. The **onstat -g ses** option displays the following information:

- The shared memory allocated for a session that is running a decision-support query
- The shared memory used by a session that is running a decision-support query
- The number of threads that the database server started for a session

For example, in Figure 12-4, session number 49 is running five threads for a decision-support query.

```
session                                          #RSAM    total      used
id       user     tty      pid   hostname threads  memory     memory
57       informix -        0     -        0        8192       5908
56       user_3   ttyp3    2318  host_10  1        65536      62404
55       user_3   ttyp3    2316  host_10  1        65536      62416
54       user_3   ttyp3    2320  host_10  1        65536      62416
53       user_3   ttyp3    2317  host_10  1        65536      62416
52       user_3   ttyp3    2319  host_10  1        65536      62416
51       user_3   ttyp3    2321  host_10  1        65536      62416
49       user_1   ttyp2    2308  host_10  5        188416     178936
2        informix -        0     -        0        8192       6780
1        informix -        0     -        0        8192       4796
```

Figure 12-4
onstat -g ses Output

Using SET EXPLAIN

When PDQ is turned on, the SET EXPLAIN output shows whether the optimizer chose parallel scans. If the optimizer chose parallel scans, the output lists `Parallel`. If PDQ is turned off, the output lists `Serial`.

If PDQ is turned on, the optimizer indicates the maximum number of threads that are required to answer the query. The output lists `# of Secondary Threads`. This field indicates the number of threads that are required in addition to your user session thread. The total number of threads necessary is the number of secondary threads plus 1.

The following example shows the SET EXPLAIN output for a table with fragmentation and PDQ priority set to LOW:

```
SELECT * FROM t1 WHERE c1 > 20

Estimated Cost: 2
Estimated # of Rows Returned: 2

1) informix.t1: SEQUENTIAL SCAN (Parallel, fragments: 2)

 Filters: informix.t1.c1 > 20

# of Secondary Threads = 1
```

The following example of SET EXPLAIN output shows a query with a hash join between two fragmented tables and PDQ priority set to ON. The query is marked with DYNAMIC HASH JOIN, and the table on which the hash is built is marked with Build Outer.

```
QUERY:
------
SELECT h1.c1, h2.c1 FROM h1, h2 WHERE h1.c1 = h2.c1

Estimated Cost: 2
Estimated # of Rows Returned: 5

1) informix.h1: SEQUENTIAL SCAN (Parallel, fragments: ALL)

2) informix.h2: SEQUENTIAL SCAN (Parallel, fragments: ALL)


DYNAMIC HASH JOIN (Build Outer)
 Dynamic Hash Filters: informix.h1.c1 = informix.h2.c1

# of Secondary Threads = 6
```

The following example of SET EXPLAIN output shows a table with fragmentation, PDQ priority set to LOW, and an index that was selected as the access plan:

```
SELECT * FROM t1 WHERE c1 < 13

Estimated Cost: 2
Estimated # of Rows Returned: 1

1) informix.t1: INDEX PATH

 (1) Index Keys: c1 (Parallel, fragments: ALL)
 Upper Index Filter: informix.t1.c1 < 13


# of Secondary Threads = 3
```

Chapter

13

Improving Individual Query Performance

In This Chapter 13-5

Using a Dedicated Test System 13-5

Displaying the Query Plan 13-6

Improving Filter Selectivity 13-7
- Filters with User-Defined Routines 13-7
- Avoiding Certain Filters 13-8
  - Avoiding Difficult Regular Expressions 13-8
  - Avoiding Noninitial Substrings 13-8
- Using Join Filters and Post-Join Filters 13-9

Updating Statistics 13-13
- Updating Number of Rows 13-14
- Dropping Data Distributions 13-15
- Creating Data Distributions 13-15
- Updating Statistics for Join Columns 13-18
- Updating Statistics for Columns with User-Defined Data Types 13-19
- Displaying Distributions 13-20
- Improving Performance of UPDATE STATISTICS 13-22

Improving Performance with Indexes 13-23
- Replacing Autoindexes with Permanent Indexes 13-23
- Using Composite Indexes 13-23
- Using Indexes for Data Warehouse Applications 13-24
- Dropping and Rebuilding Indexes After Updates 13-26

Improving Performance of Distributed Queries 13-26
Buffering Data Transfers for a Distributed Query 13-26
Displaying a Query Plan for a Distributed Query 13-27

Improving Sequential Scans 13-28

Reducing the Impact of Join and Sort Operations 13-29
Avoiding or Simplifying Sort Operations 13-29
Using Parallel Sorts 13-30
Using Temporary Tables to Reduce Sorting Scope 13-31

Optimizing User-Response Time for Queries 13-32
Optimization Level 13-32
Optimization Goal 13-33
Specifying the Query Performance Goal 13-34
Preferred Query Plans for User-Response-Time Optimization. 13-35

Optimizing Queries for User-Defined Data Types 13-37
Parallel UDRs 13-38
Selectivity and Cost Functions. 13-39
User-Defined Statistics for UDTs 13-40
Negator Functions 13-41

SQL Statement Cache 13-41
When to Use the SQL Statement Cache. 13-42
Using the SQL Statement Cache 13-43
Enabling the SQL Statement Cache. 13-44
Placing Statements in the Cache 13-46
Monitoring Memory Usage for Each Session 13-46
onstat -g ses. 13-47
onstat -g ses session-id 13-48
onstat -g sql session-id 13-49
onstat -g stm session-id. 13-49
Monitoring Usage of the SQL Statement Cache 13-50

Monitoring Sessions and Threads 13-52
Using Command-Line Utilities 13-52
onstat -u 13-52
onstat -g ath 13-54
onstat -g act 13-54
onstat -g ses 13-55
onstat -g mem and onstat -g stm 13-55
Using ON-Monitor to Monitor Sessions 13-56
Using ISA to Monitor Sessions 13-57
Using SMI Tables 13-57

Monitoring Transactions 13-58
Displaying Transactions with onstat -x 13-59
Displaying Locks with onstat -k. 13-61
Displaying User Sessions with onstat -u 13-62
Displaying Sessions Executing SQL Statements 13-63

In This Chapter

This chapter suggests ways to apply the general and conceptual information, in addition to the monitoring information, to improve the performance of a query.

Using a Dedicated Test System

If possible, you might decide to test a query on a system that does not interfere with production database servers. Even if you use your database server as a data warehouse, you might sometimes test queries on a separate system until you understand the tuning issues that are relevant to the query. However, testing queries on a separate system might distort your tuning decisions in several ways.

If you are trying to improve performance of a large query, one that might take several minutes or hours to complete, you can prepare a scaled-down database in which your tests can complete more quickly. However, be aware of these potential problems:

- The optimizer can make different choices in a small database than in a large one, even when the relative sizes of tables are the same. Verify that the query plan is the same in the real and the model databases.
- Execution time is rarely a linear function of table size. For example, sorting time increases faster than table size, as does the cost of indexed access when an index goes from two to three levels. What appears to be a big improvement in the scaled-down environment can be insignificant when applied to the full database.

Therefore, any conclusion that you reach as a result of tests in the model database must be tentative until you verify them in the production database.

You can often improve performance by adjusting your query or data model with the following goals in mind:

- If you are using a multiuser system or a network, where system load varies widely from hour to hour, try to perform your experiments at the same time each day to obtain repeatable results. Initiate tests when the system load is consistently light so that you are truly measuring the impact of your query only.
- If the query is embedded in a complicated program, you can extract the SELECT statement and embed it in a DB-Access script.

Displaying the Query Plan

Before you change a query, study its query plan to determine the kind and amount of resources it requires. The query plan shows what parallel scans are used, the maximum number of threads required, the indexes used, and so on. Then examine your data model to see if the changes this chapter suggests will improve it.

You can display the query plan with one of the following methods:

- Execute one of the following SET EXPLAIN statements just before the query:
  - SET EXPLAIN ON

    This SQL statement displays the chosen query plan. For a description of the SET EXPLAIN ON output, see "Query Plan Report" on page 10-12.
  - SET EXPLAIN ON AVOID_EXECUTE

    This SQL statement displays the chosen query plan and does not execute the query.
- Use one of the following EXPLAIN directives in the query:
  - EXPLAIN
  - EXPLAIN AVOID_EXECUTE

For more information on these EXPLAIN directives, see "EXPLAIN Directives" on page 11-16.

Improving Filter Selectivity

The greater the precision with which you specify the desired rows, the greater the likelihood that your queries will complete quickly. To control the amount of information that the query evaluates, use the WHERE clause of the SELECT statement. The conditional expression in the WHERE clause is commonly called a *filter*.

For information on how filter selectivity affects the query plan that the optimizer chooses, refer to "Filters in the Query" on page 10-23. The following sections provide some guidelines to improve filter selectivity.

Filters with User-Defined Routines

Query filters can include user-defined routines. You can improve the selectivity of filters that include UDRs with the following features:

- Functional indexes

  You can create a *functional index* on the resulting values of a user-defined routine or a built-in function that operates on one or more columns. When you create a functional index, the database server computes the return values of the function and stores them in the index. The database server can locate the return value of the function in an appropriate index without executing the function for each qualifying column.

  For more information on indexing user-defined functions, refer to "Using a Functional Index" on page 7-28.

- User-defined selectivity functions

  You can write a function that calculates the expected fraction of rows that qualify for the function. For a brief description of user-defined selectivity functions, refer to "Selectivity and Cost Functions" on page 13-39. For more information on how to write and register user-defined selectivity functions, refer to *Creating User-Defined Routines and User-Defined Data Types*.

Avoiding Certain Filters

For best performance, avoid the following types of filters:

- Certain difficult regular expressions
- Noninitial substrings

The following sections describe these types of filters and the reasons for avoiding them.

Avoiding Difficult Regular Expressions

The MATCHES and LIKE keywords support *wildcard* matches, which are technically known as *regular expressions*. Some regular expressions are more difficult than others for the database server to process. A wildcard in the initial position, as in the following example (find customers whose first names do not end in *y*), forces the database server to examine every value in the column:

```
SELECT * FROM customer WHERE fname NOT LIKE '%y'
```

You cannot use an index with such a filter, so the table in this example must be accessed sequentially.

If a difficult test for a regular expression is essential, avoid combining it with a join. If necessary, process the single table and apply the test for a regular expression to select the desired rows. Save the result in a temporary table and join that table to the others.

Regular-expression tests with wildcards in the middle or at the end of the operand do not prevent the use of an index when one exists.

Avoiding Noninitial Substrings

A filter based on a noninitial substring of a column also requires the database server to test every value in the column, as the following example shows:

```
SELECT * FROM customer
   WHERE zipcode[4,5] > '50'
```

The database server cannot use an index to evaluate such a filter.

The optimizer uses an index to process a filter that tests an initial substring of an indexed column. However, the presence of the substring test can interfere with the use of a composite index to test both the substring column and another column.

Using Join Filters and Post-Join Filters

The database server provides support for a subset of the ANSI join syntax that includes the following keywords:

- ON keyword to specify the join condition and any optional join filters
- LEFT OUTER JOIN keywords to specify which table is the dominant table (also referred to as *outer table*)

For more information on this ANSI join syntax, refer to the documentation notes for the *Informix Guide to SQL: Syntax*.

In an ANSI outer join, the database server takes the following actions to process the filters:

- Applies the join condition in the ON clause to determine which rows of the subordinate table (also referred to as *inner table*) to join to the outer table
- Applies optional join filters in the ON clause before and during the join

  If you specify a join filter on a base inner table in the ON clause, the database server can apply it prior to the join, during the scan of the data from the inner table. Filters on a base subordinate table in the ON clause can provide the following additional performance benefits:

  - Fewer rows to scan from the inner table prior to the join
  - Use of index to retrieve rows from the inner table prior to the join
  - Fewer rows to join
  - Fewer rows to evaluate for filters in the WHERE clause

  For information on what occurs when you specify a join filter on an outer table in the ON clause, refer to the documentation notes for the *Informix Guide to SQL: Syntax*.

- Applies filters in the WHERE clause after the join

  As usual, filters in the WHERE clause can reduce the number of rows that the database server needs to scan and reduce the number of rows returned to the user. This section uses the term *post-join filters* for these WHERE clause filters.

The demonstration database has the **customer** table and the **cust_calls** table that keep track of customer calls to the service department. Suppose a certain call code had many occurrences in the past, and you want to see if calls of this kind have decreased. To see if customers no longer have this call code, use an outer join to list all customers.

Figure 13-1 shows a sample SQL statement to accomplish this ANSI join query and the SET EXPLAIN ON output for it.

```
QUERY:
------
SELECT c.customer_num, c.lname, c.company,
c.phone, u.call_dtime, u.call_code, u.call_descr
FROM customer c
LEFT JOIN cust_calls u ON c.customer_num = u.customer_num
ORDER BY u.call_dtime

Estimated Cost: 14
Estimated # of Rows Returned: 29
Temporary Files Required For: Order By

1) virginia.c: SEQUENTIAL SCAN

2) virginia.u: INDEX PATH

   (1) Index Keys: customer_num call_dtime (Serial, fragments: ALL)
     Lower Index Filter: virginia.c.customer_num = virginia.u.customer_num

  ON-Filters:virginia.c.customer_num = virginia.u.customer_num
  NESTED LOOP JOIN(LEFT OUTER JOIN)
```

Figure 13-1
SET EXPLAIN ON Output for ANSI Join

Look at the following lines in the SET EXPLAIN ON output in Figure 13-1:

- The `ON-Filters:` line lists the join condition that was specified in the ON clause.
- The last line of the SET EXPLAIN ON output shows all three keywords (`LEFT OUTER JOIN`) for the ANSI join even though this query specifies only the LEFT JOIN keywords in the FROM clause. The OUTER keyword is optional.

Figure 13-2 shows the SET EXPLAIN ON output for an ANSI join with a join filter that checks for calls with the I call_code.

```
QUERY:
------
SELECT c.customer_num, c.lname, c.company,
c.phone, u.call_dtime, u.call_code, u.call_descr
FROM customer c LEFT JOIN cust_calls u
ON c.customer_num = u.customer_num
AND u.call_code = 'I'
ORDER BY u.call_dtime

Estimated Cost: 13
Estimated # of Rows Returned: 25
Temporary Files Required For: Order By

  1) virginia.c: SEQUENTIAL SCAN

  2) virginia.u: INDEX PATH

     Filters: virginia.u.call_code = 'I'

      (1) Index Keys: customer_num call_dtime   (Serial, fragments: ALL)
          Lower Index Filter: virginia.c.customer_num =
virginia.u.customer_num

ON-Filters:(virginia.c.customer_num = virginia.u.customer_num
               AND virginia.u.call_code = 'I' )
NESTED LOOP JOIN(LEFT OUTER JOIN)
```

Figure 13-2
SET EXPLAIN ON Output for Join Filter in ANSI Join

The main differences between the output in Figure 13-1 and Figure 13-2 are as follows:

- The optimizer chooses a different index to scan the inner table.

  This new index exploits more filters and retrieves a smaller number of rows. Consequently, the join operates on fewer rows.
- The ON clause join filter contains an additional filter.

The value in the `Estimated # of Rows Returned` line is only an estimate and does not always reflect the actual number of rows returned. The sample query in Figure 13-2 returns fewer rows than the query in Figure 13-1 because of the additional filter.

Figure 13-3 shows the SET EXPLAIN ON output for an ANSI join query that has a filter in the WHERE clause.

```
QUERY:
------
SELECT c.customer_num, c.lname, c.company,
   c.phone, u.call_dtime, u.call_code, u.call_descr
FROM customer c LEFT JOIN cust_calls u
ON c.customer_num = u.customer_num
   AND u.call_code = 'I'
WHERE c.zipcode = "94040"
ORDER BY u.call_dtime

Estimated Cost: 3
Estimated # of Rows Returned: 1
Temporary Files Required For: Order By

  1) virginia.c: INDEX PATH

    (1) Index Keys: zipcode   (Serial, fragments: ALL)
        Lower Index Filter: virginia.c.zipcode = '94040'

  2) virginia.u: INDEX PATH

     Filters: virginia.u.call_code = 'I'

    (1) Index Keys: customer_num call_dtime   (Serial, fragments: ALL)
        Lower Index Filter: virginia.c.customer_num = virginia.u.customer_num

ON-Filters:(virginia.c.customer_num = virginia.u.customer_num
      AND virginia.u.call_code = 'I' )
NESTED LOOP JOIN(LEFT OUTER JOIN)

PostJoin-Filters:virginia.c.zipcode = '94040'
```

Figure 13-3
SET EXPLAIN ON Output for WHERE Clause Filter in ANSI Join

The main differences between the output in Figure 13-2 and Figure 13-3 are as follows:

- The index on the **zipcode** column in the post-join filter is chosen for the dominant table.
- The `PostJoin-Filters` line shows the filter in the WHERE clause.

Updating Statistics

The UPDATE STATISTICS statement updates the statistics in the system catalogs that the optimizer uses to determine the lowest-cost query plan. For more information on the specific statistics that the database server keeps in the system catalog tables, refer to "Statistics Held for the Table and Index" on page 10-21.

To ensure that the optimizer selects a query plan that best reflects the current state of your tables, run UPDATE STATISTICS at regular intervals. The following table summarizes when to run different UPDATE STATISTICS statements and provides cross-references to subsequent sections. If you have many tables, you can write a script to generate these UPDATE STATISTICS statements. ISA can generate many of these UPDATE STATISTICS statements for your tables.

When to Execute	UPDATE STATISTICS Statement	Reference for Details and Examples	ISA Generates Statement
Number of rows has changed significantly or After migration from previous version of database server	UPDATE STATISTICS LOW or UPDATE STATISTICS LOW DROP DISTRIBUTIONS	"Updating Number of Rows" on page 13-14 "Dropping Data Distributions" on page 13-15	No
Queries have non-indexed join columns or filter columns	UPDATE STATISTICS MEDIUM DISTRIBUTIONS ONLY	"Creating Data Distributions," Step 1 on page 13-16	Yes
Queries have an indexed join columns or filter columns	UPDATE STATISTICS HIGH table (leading column in index)	"Creating Data Distributions," Step 2 on page 13-16	Yes
Queries have a multicolumn indexed defined on join columns or filter columns	UPDATE STATISTICS HIGH table (first differing column in multicolumn index)	"Creating Data Distributions," Step 3 on page 13-17	No

(1 of 2)

When to Execute	UPDATE STATISTICS Statement	Reference for Details and Examples	ISA Generates Statement
Queries have a multicolumn indexed defined on join columns or filter columns	UPDATE STATISTICS LOW table (all columns in multi-column index)	"Creating Data Distributions," Step 4 on page 13-17	No
Queries have many small tables (fit into one extent)	UPDATE STATISTICS HIGH on small tables	"Creating Data Distributions," Step 5 on page 13-17	No
Queries use SPL routines	UPDATE STATISTICS FOR PROCEDURE	"Reoptimizing SPL Routines" on page 10-40	No

(2 of 2)

Updating Number of Rows

When you run UPDATE STATISTICS LOW, the database server updates the statistics in the table, row, and page counts in the system catalog tables.

Run UPDATE STATISTICS LOW as often as necessary to ensure that the statistic for the number of rows is as current as possible. If the cardinality of a table changes often, run the statement more often for that table.

LOW is the default mode for UPDATE STATISTICS. The following sample SQL statement updates the statistics in the **systables**, **syscolumns**, and **sysindexes** system catalog tables but does not update the data distributions:

```
UPDATE STATISTICS FOR TABLE tab1;
```

Dropping Data Distributions

When you upgrade to a new version of the database server, you might need to drop distributions to remove the old distribution structure in the **sysdistrib** system catalog table.

```
UPDATE STATISTICS DROP DISTRIBUTIONS;
```

Creating Data Distributions

You can use the MEDIUM or HIGH keywords with the UPDATE STATISTICS statement to specify the mode for data distributions on specific columns. These keywords indicate that the database server is to generate statistics about the distribution of data values for each specified column and place that information in a system catalog table called **sysdistrib**. If a distribution has been generated for a column, the optimizer uses that information to estimate the number of rows that match a query against a column. Data distributions in **sysdistrib** supercede values in the **colmin** and **colmax** column of the **syscolumns** system catalog table when the optimizer estimates the number of rows returned.

When you use data-distribution statistics for the first time, try to update statistics in MEDIUM mode for all your tables and then update statistics in HIGH mode for all columns that head indexes. This strategy produces statistical query estimates for the columns that you specify. These estimates, on average, have a margin of error less than *percent* of the total number of rows in the table, where *percent* is the value that you specify in the RESOLUTION clause in the MEDIUM mode. The default percent value for MEDIUM mode is 2.5 percent. (For columns with HIGH mode distributions, the default resolution is 0.5 percent.) For each table that your query accesses, build data distributions according to the following guidelines.

To build data distributions for each table that your query accesses:

1. Run UPDATE STATISTICS MEDIUM for all columns in a table that do not head an index.

   This step is a single UPDATE STATISTICS statement. The default parameters are sufficient unless the table is very large, in which case you should use a resolution of `1.0` and confidence of `0.99`.

   With the DISTRIBUTIONS ONLY option, you can execute UPDATE STATISTICS MEDIUM at the table level or for the entire system because the overhead of the extra columns is not large. Run the following UPDATE STATISTICS statement to create distributions for non-index join columns and non-index filter columns:

   ```
   UPDATE STATISTICS MEDIUM DISTRIBUTIONS ONLY;
   ```

2. Run UPDATE STATISTICS HIGH for all columns that head an index. For the fastest execution time of the UPDATE STATISTICS statement, you must execute one UPDATE STATISTICS HIGH statement for each column.

   For example, suppose you have a table **t1** with columns **a**, **b**, **c**, **d**, **e**, and **f** with the following indexes:

   ```
   CREATE INDEX ix_1 ON t1 (a, b, c, d) ...
   CREATE INDEX ix_3 ON t1 (f) ...
   ```

   Run the following UPDATE STATISTICS statements for the columns that head an index:

   ```
   UPDATE STATISTICS HIGH FOR TABLE t1(a);
   UPDATE STATISTICS HIGH FOR TABLE t1(f);
   ```

   These UPDATE STATISTICS HIGH statements replace the medium distributions that the previous step creates with high distributions for index columns.

Important: *Always execute the MEDIUM mode of UPDATE STATISTICS before the HIGH mode on a table. If you execute UPDATE STATISTICS in HIGH mode followed by MEDIUM mode, you lose the high distributions.*

3. If you have indexes that begin with the same subset of columns, run UPDATE STATISTICS HIGH for the first column in each index that differs.

   For example, suppose you have the following indexes on table **t1:**

   ```
   CREATE INDEX ix_1 ON t1 (a, b, c, d) ...
   CREATE INDEX ix_2 ON t1 (a, b, e, f) ...
   CREATE INDEX ix_3 ON t1 (f) ...
   ```

   Step 2 executes UPDATE STATISTICS HIGH on column **a** and column **f** by. Then run UPDATE STATISTICS HIGH on columns **c** and **e**.

   ```
   UPDATE STATISTICS HIGH FOR TABLE t1(c);
   UPDATE STATISTICS HIGH FOR TABLE t1(e);
   ```

   In addition, you can run UPDATE STATISTICS HIGH on column **b**, but this step is usually not necessary.

4. For each multicolumn index, execute UPDATE STATISTICS LOW for all of its columns.

   For the single-column indexes in the preceding step, UPDATE STATISTICS LOW is implicitly executed when you execute UPDATE STATISTICS HIGH.

   For the sample indexes in the preceding step, run the following UPDATE STATISTICS statement to update the **sysindexes** and **syscolumns** system catalog tables:

   ```
   UPDATE STATISTICS FOR TABLE t1(a,b,c,d);
   UPDATE STATISTICS FOR TABLE t1(a,b,e,f);
   ```

5. For small tables, run UPDATE STATISTICS HIGH.

   ```
   UPDATE STATISTICS HIGH FOR TABLE t2;
   ```

Because the statement constructs the statistics only once for each index, these steps ensure that UPDATE STATISTICS executes rapidly.

For additional information about data distributions and the UPDATE STATISTICS statement, see the *Informix Guide to SQL: Syntax*.

Updating Statistics for Join Columns

Because of improvements and adjusted cost estimates to establish better query plans, the optimizer depends greatly on an accurate understanding of the underlying data distributions in certain cases. You might still think that a complex query does not execute quickly enough, even though you followed the guidelines in "Creating Data Distributions" on page 13-15. If your query involves equality predicates, take one of the following actions:

- Run the UPDATE STATISTICS statement with the HIGH keyword for specific join columns that appear in the WHERE clause of the query. If you followed the guidelines in "Creating Data Distributions" on page 13-15, columns that head indexes already have HIGH mode distributions.
- Determine whether HIGH mode distribution information on columns that do not head indexes can provide a better execution path, take the following steps:

To determine if UPDATE STATISTICS HIGH on join columns might make a difference

1. Issue the SET EXPLAIN ON statement and rerun the query.
2. Note the estimated number of rows in the SET EXPLAIN output and the actual number of rows that the query returns.
3. If these two numbers are significantly different, run UPDATE STATISTICS HIGH on the columns that participate in joins, unless you have already done so.

Important: *If your table is very large, UPDATE STATISTICS with the HIGH mode can take a long time to execute.*

The following example shows a query that involves join columns:

```
SELECT employee.name, address.city
   FROM employee, address
   WHERE employee.ssn = address.ssn
   AND employee.name = 'James'
```

In this example, the join columns are the **ssn** fields in the **employee** and **address** tables. The data distributions for both of these columns must accurately reflect the actual data so that the optimizer can correctly determine the best join plan and execution order.

You cannot use the UPDATE STATISTICS statement to create data distributions for a table that is external to the current database. For additional information about data distributions and the UPDATE STATISTICS statement, see the *Informix Guide to SQL: Syntax*.

Updating Statistics for Columns with User-Defined Data Types

Because information about the nature and use of a user-defined data type (UDT) is not available to the database server, it cannot collect the **colmin** and **colmax** column of the **syscolumns** system catalog table for user-defined data types. To gather statistics for columns with user-defined data types, programmers must write functions that extend the UPDATE STATISTICS statement. For more information, refer to the performance chapter in *Creating User-Defined Routines and User-Defined Data Types*.

Because the data distributions for user-defined data types can be large, you can optionally store them in an sbspace instead of the **sysdistrib** system catalog table.

To store data distributions for user-defined data types in an sbspace

1. Use the **onspaces -c -S** command to create an sbspace.

   To ensure recoverability of the data distributions, specify `LOGGING=ON` in the **-Df** option, as the following sample shows:

   ```
   % onspaces -c -S distrib_sbsp -p /dev/raw_dev1 -o 500 -s
         20000
      -m /dev/raw_dev2 500 -Ms 150 -Mo 200 -Df
         "AVG_LO_SIZE=32,LOGGING=ON"
   ```

   For information on sizing an sbspace, refer to "Estimating Pages That Smart Large Objects Occupy" on page 6-20.

   For more information about specifying storage characteristics for sbspaces, refer to "Configuration Parameters That Affect Sbspace I/O" on page 5-28.

2. Specify the sbspace that you created in step 1 in the configuration parameter SYSSBSPACENAME.

3. Specify the column with the user-defined data type when you run the UPDATE STATISTICS statement with the MEDIUM or HIGH keywords to generate data distributions.

To print the data distributions for a column with a user-defined data type, use the **dbschema -hd** option.

Displaying Distributions

Unless column values change considerably, you do not need to regenerate the data distributions. To verify the accuracy of the distribution, compare **dbschema -hd** output with the results of appropriately constructed SELECT statements.

For example, the following **dbschema** command produces a list of distributions for each column of table **customer** in database **vjp_stores** with the number of values in each bin, and the number of distinct values:

```
dbschema -hd customer -d vjp_stores
```

Figure 13-4 shows the data distributions for the column **zipcode** that this **dbschema -hd** command produces. Because this column heads the **zip_ix** index, UPDATE STATISTICS HIGH was run on it, as the following output line indicates:

```
High Mode, 0.500000 Resolution
```

Figure 13-4 shows 17 bins with one distinct **zipcode** value in each bin.

```
dbschema -hd customer -d vjp_stores

...
Distribution for virginia.customer.zipcode

Constructed on 09/18/2000

High Mode, 0.500000 Resolution

--- DISTRIBUTION ---

     (          02135 )
  1: (  1,    1, 02135 )
  2: (  1,    1, 08002 )
  3: (  1,    1, 08540 )
  4: (  1,    1, 19898 )
  5: (  1,    1, 32256 )
  6: (  1,    1, 60406 )
  7: (  1,    1, 74006 )
  8: (  1,    1, 80219 )
  9: (  1,    1, 85008 )
 10: (  1,    1, 85016 )
 11: (  1,    1, 94026 )
 12: (  1,    1, 94040 )
 13: (  1,    1, 94085 )
 14: (  1,    1, 94117 )
 15: (  1,    1, 94303 )
 16: (  1,    1, 94304 )
 17: (  1,    1, 94609 )


--- OVERFLOW ---

  1: (  2,       94022 )
  2: (  2,       94025 )
  3: (  2,       94062 )
  4: (  3,       94063 )
  5: (  2,       94086 )
```

Figure 13-4 *Displaying Data Distributions with dbschema -hd*

The OVERFLOW portion of the output shows the duplicate values that might skew the distribution data, so **dbschema** moves them from the distribution to a separate list. The number of duplicates in this overflow list must be greater than a critical amount that the following formula determines. Figure 13-4 shows a resolution value of .0050. Therefore, this formula determines that any value that is duplicated more than one time is listed in the overflow section.

```
Overflow = .25 * resolution * number_rows
      = .25 * .0050 * 28
      = .035
```

For more information on the **dbschema** utility, refer to the *Informix Migration Guide*.

Improving Performance of UPDATE STATISTICS

When you execute the UPDATE STATISTICS statement, the database server uses memory and disk to sort and construct data distributions. You can affect the amount of memory and disk available for UPDATE STATISTICS with the following methods:

- PDQ priority

  Although the UPDATE STATISTICS statement is not processed in parallel, you can obtain more memory for sorting when you set PDQ priority greater than 0. The default value for PDQ priority is 0. To set PDQ priority, use either the **PDQPRIORITY** environment variable or the SQL statement SET PDQPRIORITY.

  For more information on PDQ priority, refer to "Allocating Resources for Parallel Database Queries" on page 12-12.

- **DBUPSPACE** environment variable

  You can use the **DBUPSPACE** environment variable to constrain the amount of system disk space that the UPDATE STATISTICS statement can use to construct multiple column distributions simultaneously.

  For more information on this environment variable, refer to the *Informix Guide to SQL: Reference*.

Improving Performance with Indexes

You can often improve the performance of a query by adding or, in some cases, removing indexes. Consider using some of the methods that the following sections describe to improve the performance of a query.

Replacing Autoindexes with Permanent Indexes

If the query plan includes an *autoindex* path to a large table, take it as a recommendation from the optimizer that you can improve performance by adding an index on that column. If you perform the query occasionally, you can reasonably let the database server build and discard an index. If you perform a query regularly, you can save time by creating a permanent index.

Using Composite Indexes

The optimizer can use a composite index (one that covers more than one column) in several ways. The database server can use an index on columns **a**, **b**, and **c** (in that order) in the following ways:

- To locate a particular row

  The database server can use a composite index when the first filter is an equality filter and subsequent columns have range (<, <=, >, >=) expressions. The following examples of filters use the columns in a composite index:

  ```
  WHERE a=1
  WHERE a>=12 AND a<15
  WHERE a=1 AND b < 5
  WHERE a=1 AND b = 17 AND c >= 40
  ```

  The following examples of filters cannot use that composite index:

  ```
  WHERE b=10
  WHERE c=221
  WHERE a>=12 AND b=15
  ```

- To replace a table scan when all of the desired columns are contained within the index

  A scan that uses the index but does not reference the table is called a *key-only search*.

- To join column **a**, columns **ab**, or columns **abc** to another table
- To implement ORDER BY or GROUP BY on columns **a**, **ab**, or **abc** but not on **b**, **c**, **ac**, or **bc**

Execution is most efficient when you create a composite index with the columns in order from most to least distinct. In other words, the column that returns the highest count of distinct rows when queried with the DISTINCT keyword in the SELECT statement should come first in the composite index.

If your application performs several long queries, each of which contains ORDER BY or GROUP BY clauses, you can sometimes improve performance by adding indexes that produce these orderings without requiring a sort. For example, the following query sorts each column in the ORDER BY clause in a different direction:

```
SELECT * FROM t1 ORDER BY a, b DESC;
```

To avoid using temporary tables to sort column **a** in ascending order and column **b** in descending order, you must create a composite index on (**a**, **b** DESC) or on (**a** DESC, **b**). You need to create only one of these indexes because of the bidirectional-traverse capability of the database server. For more information on bidirectional traverse, refer to the *Informix Guide to SQL: Syntax*.

On the other hand, it can be less expensive to perform a table scan and sort the results instead of using the composite index when the following criteria are met:

- Your table is well ordered relative to your index.
- The number of rows that the query retrieves represents a large percentage of the available data.

Using Indexes for Data Warehouse Applications

Many data warehouse databases use a *star schema*. A star schema consists of a *fact* table and a number of *dimensional* tables. The fact table is generally large and contains the quantitative or factual information about the subject. A dimensional table describes an attribute in the fact table.

When a dimension needs lower-level information, the dimension is modeled by a hierarchy of tables, called a *snowflake schema*.

For more information on star schemas and snowflake schemas, refer to the *Informix Guide to Database Design and Implementation*.

Queries that use tables in a star schema or snowflake schema can benefit from the proper index on the fact table. Consider the example of a star schema with one fact table named **orders** and four dimensional tables named **customers**, **suppliers**, **products**, and **clerks**. The **orders** table describes the details of each sale order, which includes the customer ID, supplier ID, product ID, and sales clerk ID. Each dimensional table describes an ID in detail. The **orders** table is large, and the four dimensional tables are small.

The following query finds the total direct sales revenue in the Menlo Park region (postal code 94025) for hard drives supplied by the Johnson supplier:

```
SELECT sum(orders.price)
FROM orders, customers, suppliers,product,clerks
WHERE orders.custid = customers.custid
   AND customers.zipcode = 94025
   AND orders.suppid = suppliers.suppid
   AND suppliers.name = 'Johnson'
   AND orders.prodid = product.prodid
   AND product.type = 'hard drive'
   AND orders.clerkid = clerks.clerkid
   AND clerks.dept = 'Direct Sales'
```

This query uses a typical star join, in which the fact table joins with all dimensional tables on a foreign key. Each dimensional table has a selective table filter.

An optimal plan for the star join is to perform a cartesian product on the four dimensional tables and then join the result with the fact table. The following index on the fact table allows the optimizer to choose the optimal query plan:

```
CREATE INDEX ON orders(custid,suppid,prodid,clerkid)
```

Without this index, the optimizer might choose to first join the fact table with one dimensional table and then join the result with the remaining dimensional tables. The optimal plan provides better performance.

Dropping and Rebuilding Indexes After Updates

When an update transaction commits, the database server B-tree cleaner removes deleted index entries and, if necessary, rebalances the index nodes. However, depending on your application (in particular, the order in which it adds and deletes keys from the index), the structure of an index can become inefficient.

Use the **oncheck -pT** command to determine the amount of free space in each index page. If your table has relatively low update activity and a large amount of free space exists, you might want to drop and re-create the index with a larger value for FILLFACTOR to make the unused disk space available.

For more information on how the database server maintains an index tree, refer to the chapter on disk structure and storage in the *Administrator's Reference*.

Improving Performance of Distributed Queries

The optimizer assumes that access to a row from a remote database takes longer than access to a row in a local database. The optimizer estimates include the cost of retrieving the row from disk and transmitting it across the network. For an example of this higher estimated cost, refer to "Displaying a Query Plan for a Distributed Query" on page 13-27.

Buffering Data Transfers for a Distributed Query

The database server determines the size of the buffer to send and receive data to and from the remote server by the following factors:

- Row size

  The database server calculates the row size by summing the average move size (if available) or the length (from the **syscolumns** system catalog table) of the columns.

- Setting of the **FET_BUF_SIZE** environment variable on the client

  You might be able to reduce the size and number of data transfers by using the **FET_BUF_SIZE** environment variable to increase the size of the buffer that the database server uses to send and receive rows to and from the remote database server.

  The minimum buffer size is 1024 or 2048 bytes, depending on the row size. If the row size is larger than either 1024 or 2048 bytes, the database server uses the **FET_BUF_SIZE** value.

  For more information on the **FET_BUF_SIZE** environment variable, refer to the *Informix Guide to SQL: Reference*.

Displaying a Query Plan for a Distributed Query

Figure 13-5 shows the chosen query plan for the distributed query.

```
QUERY:
------
select l.customer_num, l.lname, l.company,
        l.phone, r.call_dtime, r.call_descr
       from customer l, vjp_stores@gilroy:cust_calls r
        where l.customer_num = r.customer_num

 Estimated Cost: 9
 Estimated # of Rows Returned: 7

   1) informix.r: REMOTE PATH

   2) informix.l: INDEX PATH

     (1) Index Keys: customer_num   (Serial, fragments: ALL)
         Lower Index Filter: informix.l.customer_num = informix.r.customer_num
 NESTED LOOP JOIN
```

Figure 13-5
Output of SET EXPLAIN ALL for Distributed Query, Part 3

The following table shows the main differences between the chosen query plans for the distributed join and the local join.

Output Line in Figure 13-5 for Distributed Query	Output Line in Figure 11-3 for Local-Only Query	Description of Difference
`vjp_stores@gilroy: virginia.cust_calls`	`informix.cust_calls`	The remote table name is prefaced with the database and server names.
`Estimated Cost: 9`	`Estimated Cost: 7`	The optimizer estimates a higher cost for the distributed query.
`informix.r: REMOTE PATH`	`informix.r: SEQUENTIAL SCAN`	The optimizer chose to keep the outer, remote **cust_calls** table at the remote site.
`select x0.call_dtime,x0.call_descr, x0.customer_num from vjp_stores:"virginia".cust_ calls x0`		The SQL statement that the local database server sends to the remote site. The remote site reoptimizes this statement to choose the actual plan.

Improving Sequential Scans

To improve performance of sequential read operations on large tables, eliminate repeated sequential scans.

Sequential access to a table other than the first table in the plan is ominous because it threatens to read every row of the table once for every row selected from the preceding tables. You should be able to judge how many times that is: perhaps a few, but perhaps hundreds or even thousands.

If the table is small, it is harmless to read it repeatedly because the table resides completely in memory. Sequential search of an in-memory table can be faster than searching the same table through an index, especially if maintaining those index pages in memory pushes other useful pages out of the buffers.

When the table is larger than a few pages, however, repeated sequential access produces poor performance. One way to prevent this problem is to provide an index to the column that is used to join the table.

Any user with the Resource privilege can build additional indexes. Use the CREATE INDEX statement to make an index.

An index consumes disk space proportional to the width of the key values and the number of rows. (See "Estimating lndex Pages" on page 7-3.) Also, the database server must update the index whenever rows are inserted, deleted, or updated; the index update slows these operations. If necessary, you can use the DROP INDEX statement to release the index after a series of queries, which frees space and makes table updates easier.

Reducing the Impact of Join and Sort Operations

After you understand what the query is doing, look for ways to obtain the same output with less effort. The following suggestions can help you rewrite your query more efficiently:

- Avoid or simplify sort operations.
- Use parallel sorts.
- Use temporary tables to reduce sorting scope.

Avoiding or Simplifying Sort Operations

Sorting is not necessarily a liability. The sort algorithm is highly tuned and extremely efficient. It is as fast as any external sort program that you might apply to the same data. You need not avoid infrequent sorts or sorts of relatively small numbers of output rows.

Try to avoid or reduce the scope of repeated sorts of large tables. The optimizer avoids a sort step whenever it can use an index to produce the output in its proper order automatically. The following factors prevent the optimizer from using an index:

- One or more of the ordered columns is not included in the index.
- The columns are named in a different sequence in the index and the ORDER BY or GROUP BY clause.
- The ordered columns are taken from different tables.

For another way to avoid sorts, see "Using Temporary Tables to Reduce Sorting Scope" on page 13-31.

If a sort is necessary, look for ways to simplify it. As discussed in "Sort-Time Costs" on page 10-28, the sort is quicker if you can sort on fewer or narrower columns.

Using Parallel Sorts

When you cannot avoid sorting, the database server takes advantage of multiple CPU resources to perform the required sort-and-merge operations in parallel. The database server can use parallel sorts for any query; parallel sorts are not limited to PDQ queries. To control the number of threads that the database server uses to sort rows, use the **PSORT_NPROCS** environment variable.

When PDQ priority is greater than 0 and **PSORT_NPROCS** is greater than 1, the query benefits both from parallel sorts and from PDQ features such as parallel scans and additional memory. Users can use the **PDQPRIORITY** environment variable to request a specific proportion of PDQ resources for a query. You can use the MAX_PDQPRIORITY parameter to limit the number of such user requests. For more information, refer to "MAX_PDQPRIORITY" on page 3-16.

In some cases, the amount of data being sorted can overflow the memory resources allocated to the query, resulting in I/O to a dbspace or sort file. For more information, refer to "Configuring Dbspaces for Temporary Tables and Sort Files" on page 5-12.

Using Temporary Tables to Reduce Sorting Scope

Building a temporary, ordered subset of a table can speed up a query. It can help to avoid multiple-sort operations and can simplify the work of the optimizer in other ways.

For example, suppose your application produces a series of reports on customers who have outstanding balances, one report for each major postal area, ordered by customer name. In other words, a series of queries occurs, each of the following form (using hypothetical table and column names):

```
SELECT cust.name, rcvbles.balance, ...other columns...
   FROM cust, rcvbles
   WHERE cust.customer_id = rcvbles.customer_id
      AND rcvbls.balance > 0
      AND cust.postcode LIKE '98_ _ _'
   ORDER BY cust.name
```

This query reads the entire **cust** table. For every row with the specified postal code, the database server searches the index on **rcvbles.customer_id** and performs a nonsequential disk access for every match. The rows are written to a temporary file and sorted. For more information on temporary files, refer to "Configuring Dbspaces for Temporary Tables and Sort Files" on page 5-12.

This procedure is acceptable if the query is performed only once, but this example includes a series of queries, each incurring the same amount of work.

An alternative is to select all customers with outstanding balances into a temporary table, ordered by customer name, as the following example shows:

```
SELECT cust.name, rcvbles.balance, ...other columns...
   FROM cust, rcvbles
   WHERE cust.customer_id = rcvbles.customer_id
      AND cvbls.balance > 0
   INTO TEMP cust_with_balance
```

You can then execute queries against the temporary table, as the following example shows:

```
SELECT *
   FROM cust_with_balance
   WHERE postcode LIKE '98_ _ _'
   ORDER BY cust.name
```

Each query reads the temporary table sequentially, but the table has fewer rows than the primary table.

Optimizing User-Response Time for Queries

The following sections describe how you can influence the time to optimize a query and the time it takes to return rows to a user.

Optimization Level

You normally obtain optimum overall performance with the default optimization level, HIGH. The time that it takes to optimize the statement is usually unimportant. However, if experimentation with your application reveals that your query is still taking too long, you can set the optimization level to LOW and then check the SET EXPLAIN output to see if the optimizer chose the same query plan as before.

To specify a HIGH or LOW level of database server optimization, use the SET OPTIMIZATION statement. For a detailed description of this statement, see the *Informix Guide to SQL: Syntax*.

Optimization Goal

The two types of optimization goals for query performance are as follows:

- Optimizing total query time
- Optimizing user-response time

Total query time is the time it takes to return all rows to the application. Total query time is most important for batch processing or for queries that require all rows be processed before returning a result to the user, as in the following query:

```
SELECT count(*) FROM orders
WHERE order_amount > 2000;
```

User-response time is the time that it takes for the database server to return a screen full of rows back to an interactive application. In interactive applications, only a screen full of data can be requested at one time. For example, the user application can display only 10 rows at one time for the following query:

```
SELECT * FROM orders
WHERE order_amount > 2000;
```

Which optimization goal is more important can have an effect on the query path that the optimizer chooses. For example, the optimizer might choose a nested-loop join instead of a hash join to execute a query if user-response time is most important, even though a hash join might result in a reduction in total query time.

Specifying the Query Performance Goal

The default behavior is for the optimizer to choose query plans that optimize the total query time. You can specify optimization of user-response time at several different levels:

- For the database server system

  To optimize user-response time, set the OPT_GOAL configuration parameter to `0`, as in the following example:

  ```
  OPT_GOAL 0
  ```

  Set OPT_GOAL to `-1` to optimize total query time.

- For the user environment

  The **OPT_GOAL** environment variable can be set before the user application starts.

  UNIX

  To optimize user-response time, set the **OPT_GOAL** environment variable to `0`, as in the following sample commands:

  ```
  Bourne shell   OPT_GOAL = 0
                 export OPT_GOAL

  C shell        setenv OPT_GOAL 0
  ```

  ♦

  For total-query-time optimization, set the **OPT_GOAL** environment variable to `-1`.

- Within the session

  You can control the optimization goal with the SET OPTIMIZATION statement in SQL. The optimization goal set with this statement stays in effect until the session ends or until another SET OPTIMIZATION statement changes the goal.

  The following statement causes the optimizer to choose query plans that favor total-query-time optimization:

  ```
  SET OPTIMIZATION ALL_ROWS
  ```

  The following statement causes the optimizer to choose query plans that favor user-response-time optimization:

  ```
  SET OPTIMIZATION FIRST_ROWS
  ```

- For individual queries

  You can use FIRST_ROWS and ALL_ROWS optimizer directives to instruct the optimizer which query goal to use. For more information about these directives, refer to "Optimization-Goal Directives" on page 11-11.

The precedence for these levels is as follows:

- Optimizer directives
- SET OPTIMIZATION statement
- **OPT_GOAL** environment variable
- OPT_GOAL configuration parameter

For example, optimizer directives take precedence over the goal that the SET OPTIMIZATION statement specifies.

Preferred Query Plans for User-Response-Time Optimization

When the optimizer chooses query plans to optimize user-response time, it computes the cost to retrieve the first row in the query for each plan and chooses the plan with the lowest cost. In some cases, the query plan with the lowest cost to retrieve the first row might not be the optimal path to retrieve all rows in the query.

The following sections explain some of the possible differences in query plans.

Nested-Loop Joins Versus Hash Join

Hash joins generally have a higher cost to retrieve the first row than nested-loop joins do. The database server must build the hash table before it retrieves any rows. However, in some cases, total query time is faster if the database server uses a hash join.

In the following example, **tab2** has an index on **col1**, but **tab1** does not have an index on **col1**. When you execute SET OPTIMIZATION ALL_ROWS before you run the query, the database server uses a hash join and ignores the existing index, as the following SET EXPLAIN output shows:

```
QUERY:
------
SELECT * FROM tab1,tab2
WHERE tab1.col1 = tab2.col1
Estimated Cost: 125
Estimated # of Rows Returned: 510
1) lsuto.tab2: SEQUENTIAL SCAN
2) lsuto.tab1: SEQUENTIAL SCAN
DYNAMIC HASH JOIN
    Dynamic Hash Filters: lsuto.tab2.col1 = lsuto.tab1.col1
```

However, when you execute SET OPTIMIZATION FIRST_ROWS before you run the query, the database server uses a nested-loop join. The clause (FIRST_ROWS OPTIMIZATION) in the following SET EXPLAIN output shows that the optimizer used user-response-time optimization for the query:

```
QUERY:        (FIRST_ROWS OPTIMIZATION)
------
SELECT * FROM tab1,tab2
WHERE tab1.col1 = tab2.col1
Estimated Cost: 145
Estimated # of Rows Returned: 510
1) lsuto.tab1: SEQUENTIAL SCAN
2) lsuto.tab2: INDEX PATH
    (1) Index Keys: col1
        Lower Index Filter: lsuto.tab2.col1 = lsuto.tab1.col1
NESTED LOOP JOIN
```

Table Scans Versus Index Scans

In cases where the database server returns a large number of rows from a table, the lower cost option for the total-query-time goal might be to scan the table instead of using an index. However, to retrieve the first row, the lower cost option for the user-response-time goal might be to use the index to access the table.

Ordering with Fragmented Indexes

When an index is not fragmented, the database server can use the index to avoid a sort. For more information on avoiding sorts, refer to "Avoiding or Simplifying Sort Operations" on page 13-29. However, when an index is fragmented, the ordering can be guaranteed only within the fragment, not between fragments.

Usually, the least expensive option for the total-query-time goal is to scan the fragments in parallel and then use the parallel sort to produce the proper ordering. However, this option does not favor the user-response-time goal.

Instead, if the user-response time is more important, the database server reads the index fragments in parallel and merges the data from all of the fragments. No additional sort is generally needed.

Optimizing Queries for User-Defined Data Types

Queries that access user-defined data types (UDTs) can take advantage of the same performance features that built-in data types use:

- Indexes

  If a query accesses a small number of rows, an index speeds retrieval because the database server does not need to read every page in a table to find the rows. For more information, refer to "Indexes on User-Defined Data Types" on page 7-22.

- Parallel database query (PDQ)

  Queries that access user-defined data can take advantage of parallel scans and parallel execution. For information about parallel execution of user-defined routines, refer to the performance chapter in *Creating User-Defined Routines and User-Defined Data Types*.

  To turn on parallel execution for a query, set the **PDQPRIORITY** environment variable or use the SQL statement SET PDQPRIORITY. For more information about how to set PDQ priority and configuration parameters that affect PDQ, refer to "Allocating Resources for Parallel Database Queries" on page 12-12.

- Optimizer directives

In addition, programmers can write the following functions or UDRs to help the optimizer create an efficient query plan for your queries:

- Parallel UDRs that can take advantage of parallel database queries
- User-defined selectivity functions that calculate the expected fraction of rows that qualify for the function
- User-defined cost functions that calculate the expected relative cost to execute a user-defined routine
- User-defined statistical functions that the UPDATE STATISTICS statement can use to generate statistics and data distributions
- User-defined negator functions to allow more choices for the optimizer

The following sections summarize these techniques. For a more complete description of how to write and register user-defined selectivity functions and user-defined cost functions, refer to *Creating User-Defined Routines and User-Defined Data Types*.

Parallel UDRs

One way to execute UDRs is in an expression in a query. You can take advantage of parallel execution if the UDR is in an expression in one of the following parts of a query:

- WHERE clause
- SELECT list
- GROUP BY list
- Overloaded comparison operator
- User-defined aggregate
- HAVING clause
- Select list for a parallel insertion statement
- Generic B-tree index scan on multiple index fragments provided that the compare function used in the B-tree index scan is parallelizable

For example, suppose you create an opaque data type **circle**, a table **cir_t** that defines a column of type **circle**, a user-defined routine **area**, and then execute the following sample query:

```
SELECT circle, area(circle)
   FROM cir_t
   WHERE circle > "(6,2,4)";
```

In this sample query, the following operations can execute in parallel:

- The UDR **area(circle)** in the SELECT list

  If the table **cir_t** is fragmented, multiple **area** UDRs can execute in parallel, one UDR on each fragment.

- The expression `circle > "(6,2,4)"` in the WHERE clause

  If the table **cir_t** is fragmented, multiple scans of the table can execute in parallel, one scan on each fragment. Then multiple ">" comparison operators can execute in parallel, one operator per fragment.

By default, a UDR does not execute in parallel. To enable parallel execution of UDRs, you must take the following actions:

- Specify the PARALLELIZABLE modifier in the CREATE FUNCTION or ALTER FUNCTION statement.
- Ensure that the UDR does not call functions that are not PDQ thread-safe.
- Turn on PDQ priority.
- Use the UDR in a parallel database query.

Selectivity and Cost Functions

The CREATE FUNCTION command allows users to create UDRs. You can place a UDR in an SQL statement, as in the following example:

```
SELECT * FROM image
WHERE get_x1(image.im2) and get_x2(image.im1)
```

The optimizer cannot accurately evaluate the cost of executing a UDR without additional information. You can provide the cost and selectivity of the function to the optimizer. The database server uses cost and selectivity together to determine the best path. For more information on selectivity, refer to "Filters with User-Defined Routines" on page 13-7.

In the previous example, the optimizer cannot determine which function to execute first, the **get_x1** function or the **get_x2** function. If a function is expensive to execute, the DBA can assign the function a larger cost or selectivity, which can influence the optimizer to change the query plan for better performance. In the previous example, if **get_x1** costs more to execute, the DBA can assign a higher cost to the function, which can cause the optimizer to execute the **get_x2** function first.

You can add the following routine modifiers to the CREATE FUNCTION statement to change the cost or selectivity that the optimizer assigns to the function:

- **selfunc**=*function_name*
- **selconst**=*integer*
- **costfunc**=*function_name*
- **percall_cost**=*integer*

For more information on cost or selectivity modifiers, refer to *Creating User-Defined Routines and User-Defined Data Types*.

User-Defined Statistics for UDTs

Because information about the nature and use of a user-defined data type (UDT) is not available to the database server, it cannot collect distributions or the **colmin** and **colmax** information (found in the **syscolumns** system catalog table) for a UDT. Instead, you can create a special function that populates these statistics. The database server runs the statistics collection function when you execute UPDATE STATISTICS.

For more information on the importance of updating statistics, refer to "Statistics Held for the Table and Index" on page 10-21. For information on improving performance, refer to "Updating Statistics for Columns with User-Defined Data Types" on page 13-19.

Negator Functions

A *negator function* takes the same arguments as its companion function, in the same order, but returns the Boolean complement. That is, if a function returns TRUE for a given set of arguments, its negator function returns FALSE when passed the same arguments, in the same order. In certain cases, the database server can process a query more efficiently if the sense of the query is reversed. That is, "Is **x** greater than **y**?" changes to "Is **y** less than or equal to **x**?"

SQL Statement Cache

Before the database server executes an SQL statement, it must first parse and optimize the statement. These steps can be time consuming, depending on the size of the SQL statement.

The database server can store the parsed and optimized SQL statement in the virtual portion of shared memory, in an area called the *SQL statement cache*. The SQL statement cache (SSC) can be accessed by all users, and it allows users to bypass the parse and optimize steps before they execute the query. This capability can result in the following significant performance improvements:

- Reduced response times when users are executing the same SQL statements.

  SQL statements that take longer to optimize (usually because they include many tables and many filters in the WHERE clause) execute faster from the SQL statement cache because the database server does not have to parse or optimize the statement.
- Reduced memory usage because the database server shares query data structures among users.

  Memory reduction with the SQL statement cache is greater when a statement has many column names in the select list.

For more information about the effect of the SQL statement cache on the performance of the overall system, refer to "SQL Statement Cache" on page 4-37.

When to Use the SQL Statement Cache

Applications might benefit from use of the SQL statement cache if multiple users execute the same SQL statements. The database server considers statements to be the same if all characters match exactly. For example, if 50 sales representatives execute the **add_order** application throughout the day, they all execute the same SQL statements if the application contains SQL statements that use host variables, such as the following example:

```
SELECT * FROM ORDERS WHERE order_num = :hostvar
```

This kind of application benefits from use of the SQL statement cache because users are likely to find the SQL statements in the SQL statement cache.

The database server does not consider the following SQL statements exact matches because they contain different literal values in the WHERE clause:

```
SELECT * FROM customer, orders
   WHERE customer.customer_num = orders.customer_num
   AND order_date > "01/01/97"
SELECT * FROM customer, orders
   WHERE customer.customer_num = orders.customer_num
   AND order_date > "01/01/1997"
```

Performance does not improve with the SQL statement cache in the following situations:

- If a report application is run once nightly, and it executes SQL statements that no other application uses, it does not benefit from use of the statement cache.
- If an application prepares a statement and then executes it many times, performance does not improve with the SQL statement cache because the statement is optimized just once during the PREPARE statement.

When a statement contains host variables, the database server replaces the host variables with placeholders when it stores the statement in the SQL statement cache. Therefore, the statement is optimized without the database server having access to the values of the host variables. In some cases, if the database server had access to the values of the host variables, the statement might be optimized differently, usually because the distributions stored for a column inform the optimizer exactly how many rows pass the filter.

If an SQL statement that contains host variables performs poorly with the SQL statement cache turned on, try flushing the SQL statement cache with the **onmode -e flush** command and running the query with values that are more frequently used across multiple executions of the query. When you flush the cache, the database server reoptimizes the query and generates a query plan that is optimized for these frequently used values.

Important: *The database server flushes an entry from the SQL statement cache only if it is not in use. If an application prepares the statement and keeps it, the entry is still in use. In this case, the application needs to close the statement before the flush is beneficial.*

Using the SQL Statement Cache

The DBA usually makes the decision to enable the SQL statement cache. If the SQL statement cache is enabled, individual users can decide whether or not to use the SQL statement cache for their specific environment or application.

The database server incurs some processing overhead in managing the SQL statement cache, so a user should not use the SQL statement cache if no other users share the SQL statements in the application.

The following section describes how the DBA can enable the SQL statement cache in one of two modes:

- Always use the SQL statement cache unless a user explicitly specifies do not use the cache.
- Use the SQL statement cache only when a user explicitly specifies use it.

Enabling the SQL Statement Cache

The database server does not use the SQL statement cache when the STMT_CACHE configuration parameter is `0` (the default value).

Use one of the following methods to change this STMT_CACHE default value:

- Update the ONCONFIG file to specify the STMT_CACHE configuration parameter and restart the database server.

  If you set the STMT_CACHE configuration parameter to `1`, the database server uses the SQL statement cache for an individual user when the user sets the **STMT_CACHE** environment variable to `1` or executes the SET STATEMENT CACHE ON statement within an application.

  ```
  STMT_CACHE 1
  ```

  If the STMT_CACHE configuration parameter is `2`, the database server stores SQL statements for all users in the SQL statement cache except when individual users turn off the feature with the **STMT_CACHE** environment variable or the SET STATEMENT CACHE OFF statement.

  ```
  STMT_CACHE 2
  ```

- Use the **onmode -e** command to override the STMT_CACHE configuration parameter dynamically.

  If you use the **enable** keyword, the database server uses the SQL statement cache for an individual user when the user sets the **STMT_CACHE** environment variable to `1` or executes the SET STATEMENT CACHE ON statement within an application.

  ```
  onmode -e enable
  ```

  If you use the **on** keyword, the database server stores SQL statements for all users in the SQL statement cache except when individual users turn off the feature with the **STMT_CACHE** environment variable or the SET STATEMENT CACHE OFF statement.

  ```
  onmode -e on
  ```

The following table summarizes the use of the SQL statement cache for a user, depending on the setting of the STMT_CACHE configuration parameter (or the execution of **onmode -e**) and the use in an application of the **STMT_CACHE** environment variable and the SET STATEMENT CACHE statement.

STMT_CACHE Configuration Parameter or onmode -e	STMT_CACHE Environment Variable	SET STATEMENT CACHE Statement	Resulting Behavior
0 (default)	Not applicable	Not applicable	Statement cache not used
1	0 (or not set)	OFF	Statement cache not used
1	1	OFF	Statement cache not used
1	0 (or not set)	ON	Statement cache used
1	1	ON	Statement cache used
1	1	Not executed	Statement cache used
1	0	Not executed	Statement cache not used
2	1 (or not set)	ON	Statement cache used
2	1 (or not set)	OFF	Statement cache not used
2	0	ON	Statement cache used
2	0	OFF	Statement cache not used by user
2	0	Not executed	Statement cache not used by user
2	1 (or not set)	Not executed	Statement cache used by user

Placing Statements in the Cache

SELECT, UPDATE, INSERT and DELETE statements can be placed in the SQL statement cache, with some exceptions. When the database server checks if an SQL statement is in the cache, it must find an exact match.

For a complete list of the exceptions and a list of requirements for an exact match, refer to SET STATEMENT CACHE in the *Informix Guide to SQL: Syntax*.

Monitoring Memory Usage for Each Session

You can use arguments of the **onstat -g** option to obtain memory information for each session.

To identify SQL statements using large amount of memory

1. Use the **onstat -u** option to view all user threads
2. Use the **onstat -g ses** option to view memory of all sessions and see which session has the highest memory usage.
3. Use the **onstat -g ses** *session-id* option to view more details on the session with the highest memory usage.
4. Use the **onstat -g stm** *session-id* option to view the memory used by the SQL statements

onstat -g ses

The **onstat -g ses** displays memory usage by session id. When the session shares the memory structures in the SSC, the value in the **used memory** column should be lower than when the cache is turned off. For example, Figure 13-6 shows sample **onstat -g ses** output when the SQL statement cache is not enabled, and Figure 13-7 shows output after it is enabled and the queries in Session 4 are run again. Figure 13-6 shows that Session 4 has 45656 bytes of used memory. Figure 13-7 shows that Session 4 has less used bytes (36920) when the SSC is enabled.

```
session                                          #RSAM    total      used
id       user     tty      pid      hostname    threads  memory     memory
12       informix -        0        -           0        12288      7632
4        informix 11       5158     smoke       1        53248      45656
3        informix -        0        -           0        12288      8872
2        informix -        0        -           0        12288      7632
```

Figure 13-6
onstat -g ses Output when the SQL Statement Cache Is Not Enabled

```
session                                          #RSAM    total      used
id       user     tty      pid      hostname    threads  memory     memory
17       informix -        0        -           0        12288      7632
16       informix 12       5258     smoke       1        40960      38784
4        informix 11       5158     smoke       1        53248      36920
3        informix -        0        -           0        12288      8872
2        informix -        0        -           0        12288      7632
```

Figure 13-7
onstat -g ses Output when the SQL Statement Cache Is Enabled

Figure 13-7 also shows the memory allocated and used for Session 16, which executes the same SQL statements as Session 4. Session 16 allocates less total memory (40960) and uses less memory (38784) than Session 4 (Figure 13-6 shows 53248 and 45656, respectively) because it uses the existing memory structures in the SQL statement cache.

onstat -g ses session-id

The **onstat -g ses** *session-id* option displays detailed information for a session. The following **onstat -g ses** *session-id* output columns display memory usage:

- The `Memory pools` portion of the output
  - The **totalsize** column shows the number of bytes currently allocated
  - The **freesize** column shows the number of unallocated bytes
- The last line of the output shows the number of bytes allocated from the sscpool.

Figure 13-8 shows that Session 16 has currently allocated 69632 bytes, of which 11600 bytes are allocated from the sscpool.

```
onstat -g ses 14


session                                        #RSAM    total      used
id       user     tty      pid      hostname  threads  memory     memory
14       virginia 7        28734    lyceum    1        69632      67384

tid      name     rstcb    flags    curstk    status
38       sqlexec  a3974d8  Y--P---  1656      cond wait(netnorm)

Memory pools    count 1
name         class addr     totalsize freesize #allocfrag #freefrag
14           V     a974020  69632     2248     156        2

...
Sess  SQL            Current            Iso Lock       SQL  ISAM F.E.
Id    Stmt type      Database           Lvl Mode       ERR  ERR  Vers
14    SELECT         vjp_stores         CR  Not Wait   0    0    9.03

Current statement name : slctcur

Current SQL statement :
  SELECT C.customer_num, O.order_num FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num AND O.order_num = I.order_num

Last parsed SQL statement :
  SELECT C.customer_num, O.order_num FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num AND O.order_num = I.order_num

 11600 byte(s) of memory is allocated from the sscpool
```

Figure 13-8
onstat -g ses session-id Output

onstat -g sql session-id

The **onstat -g sql** *session-id* option displays information about the SQL statements executed by the session. Figure 13-9 shows that **onstat -g sql** *session-id* displays the same information as the bottom portion of the **onstat -g ses** *session-id* in Figure 13-8, which includes the number of bytes allocated from the sscpool.

```
onstat -g sql 14


Sess  SQL            Current            Iso Lock       SQL  ISAM F.E.
Id    Stmt type      Database           Lvl Mode       ERR  ERR  Vers
14    SELECT         vjp_stores         CR  Not Wait   0    0    9.03

Current statement name : slctcur

Current SQL statement :
  SELECT C.customer_num, O.order_num FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num AND O.order_num = I.order_num

Last parsed SQL statement :
  SELECT C.customer_num, O.order_num FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num AND O.order_num = I.order_num

 11600 byte(s) of memory is allocated from the sscpool
```

Figure 13-9
onstat -g sql session-id Output

onstat -g stm session-id

The **onstat -g stm** *session-id* option displays information about the memory each SQL statement uses in a session. Figure 13-10 displays the output of **onstat -g stm** *session-id* for the same session (14) as in **onstat -g ses** *session-id* in Figure 13-8 on page 13-48 and **onstat -g sql** *session-id* in Figure 13-9 on page 13-49. When the SQL statement cache is on, the database server creates the heaps in the sscpool. Therefore, the **heapsz** output field in Figure 13-10 shows that this SQL statement uses 10056 bytes, which is contained within the 11600 bytes in the sscpool that the **onstat -g sql 14** shows.

```
onstat -g stm 14


session   14 ---------------------------------------------------------------
 sdblock  heapsz  statement ('*' = Open cursor)
 aa11018   10056 *SELECT C.customer_num, O.order_num
FROM customer C, orders O, items I
WHERE C.customer_num = O.customer_num
AND O.order_num = I.order_num
```

Figure 13-10
onstat -g stm session-id Output

Monitoring Usage of the SQL Statement Cache

If you notice a sudden increase in response time for a query that had been using the SQL statement cache, the entry might have been dropped or deleted. The database server drops an entry from the cache when one of the objects that the query depends on is altered so that it invalidates the data dictionary cache entry for the query. The following operations cause a dependency check failure:

- Execution of any data definition language (DDL) statement (such as ALTER TABLE, DROP INDEX, or CREATE INDEX) that might alter the query plan
- Alteration of a table that is linked to another table with a referential constraint (in either direction)
- Execution of UPDATE STATISTICS FOR TABLE for any table or column involved in the query
- Renaming a column, database, or index with the RENAME statement

When an entry is marked as dropped or deleted, the database server must reparse and reoptimize the SQL statement the next time it executes. For example, Figure 13-11 on page 13-51 shows the entries that **onstat -g ssc** displays after UPDATE STATISTICS was executed on the **items** and **orders** table between the execution of the first and second SQL statements.

The `Statement Cache Entries:` portion of the **onstat -g ssc** output in Figure 13-11 displays a **flag** field that indicates whether or not an entry has been dropped or deleted from the SQL statement cache.

- The first entry has a **flag** column with the value `DF`, which indicates that the entry is fully cached, but is now dropped because its entry was invalidated.
- The second entry has the same statement text as the third entry, which indicates that it was reparsed and reoptimized when it was executed after the UPDATE STATISTICS statement.

```
onstat -g ssc

...
Statement Cache Entries:

lru hash ref_cnt hits flag heap_ptr  database     user
-------------------------------------------------------------------------------
...
  2  232       1    1   DF  aa3d020  vjp_stores   virginia
  SELECT C.customer_num, O.order_num
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
    AND O.order_num = I.order_num

  3  232       1    0   -F  aa8b020  vjp_stores   virginia
  SELECT C.customer_num, O.order_num
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
    AND O.order_num = I.order_num
...
```

Figure 13-11
Sample onstat -g ssc Output for Dropped Entry

Monitoring Sessions and Threads

To monitor database server activity, you can view the number of active sessions and threads and the amount of resources that they are using. Monitoring sessions and threads is important for sessions that perform queries as well as sessions that perform inserts, updates, and deletes. Some of the information that you can monitor for sessions and threads allows you to determine if an application is using a disproportionate amount of the resources.

Using Command-Line Utilities

Use the following options of the **onstat** utility to monitor sessions and threads:

- **onstat -u**
- **onstat -g ath**
- **onstat -a act**
- **onstat -a ses**
- **onstat -g mem**
- **onstat -g stm**

onstat -u

The **onstat -u** option displays information on active threads that require a database server task-control block. Active threads include threads that belong to user sessions, as well as some that correspond to database server daemons (for example, page cleaners). Figure 13-12 on page 13-53 shows an example of output from this utility.

Use **onstat -u** to determine if a user is waiting for a resource or holding too many locks, or to get an idea of how much I/O the user has performed.

The utility output displays the following information:

- The address of each thread
- Flags that indicate the present state of the thread (for example, waiting for a buffer or waiting for a checkpoint), whether the thread is the primary thread for a session, and what type of thread it is (for example, user thread, daemon thread, and so on)

  For information on these flags, refer to the *Administrator's Reference*.
- The session ID and user login ID for the session to which the thread belongs

  A session ID of `0` indicates a daemon thread.
- Whether the thread is waiting for a specific resource and the address of that resource
- The number of locks that the thread is holding
- The number of read calls and the number of write calls that the thread has executed
- The maximum number of concurrent user threads that were allocated since you last initialized the database server

If you execute **onstat -u** while the database server is performing fast recovery, several database server threads might appear in the display.

```
Userthreads
address  flags   sessid   user     tty      wait     tout locks nreads
nwrites
80eb8c   ---P--D 0        informix -        0        0    0     33       19
80ef18   ---P--F 0        informix -        0        0    0     0        0
80f2a4   ---P--B 3        informix -        0        0    0     0        0
80f630   ---P--D 0        informix -        0        0    0     0        0
80fd48   ---P--- 45       chrisw   ttyp3    0        0    1     573      237
810460   ------- 10       chrisw   ttyp2    0        0    1     1        0
810b78   ---PR-- 42       chrisw   ttyp3    0        0    1     595      243
810f04   Y------ 10       chrisw   ttyp2    beacf8   0    1     1        0
811290   ---P--- 47       chrisw   ttyp3    0        0    2     585      235
81161c   ---PR-- 46       chrisw   ttyp3    0        0    1     571      239
8119a8   Y------ 10       chrisw   ttyp2    a8a944   0    1     1        0
81244c   ---P--- 43       chrisw   ttyp3    0        0    2     588      230
8127d8   ----R-- 10       chrisw   ttyp2    0        0    1     1        0
812b64   ---P--- 10       chrisw   ttyp2    0        0    1     20       0
812ef0   ---PR-- 44       chrisw   ttyp3    0        0    1     587      227
 15 active, 20 total, 17 maximum concurrent
```

Figure 13-12
onstat -u Output

onstat -g ath

Use the **onstat -g ath** option to obtain a listing of all threads. Unlike the **onstat -u** option, this listing includes internal daemon threads that do not have a database server task-control block. On the other hand, the **onstat -g ath** display does not include the session ID (because not all threads belong to sessions).

Threads that a primary decision-support thread started have a name that indicates their role in the decision-support query. For example, Figure 13-13 shows four scan threads that belong to a decision-support thread.

```
Threads:
tid     tcb     rstcb   prty    status                  vp-class   name
...
11      994060   0        4     sleeping(Forever)         1cpu    kaio
12      994394   80f2a4   2     sleeping(secs: 51)        1cpu    btclean
26      99b11c   80f630   4     ready                     1cpu    onmode_mon
32      a9a294   812b64   2     ready                     1cpu    sqlexec
113     b72a7c   810b78   2     ready                     1cpu    sqlexec
114     b86c8c   81244c   2     cond wait(netnorm)        1cpu    sqlexec
115     b98a7c   812ef0   2     cond wait(netnorm)        1cpu    sqlexec
116     bb4a24   80fd48   2     cond wait(netnorm)        1cpu    sqlexec
117     bc6a24   81161c   2     cond wait(netnorm)        1cpu    sqlexec
118     bd8a24   811290   2     ready                     1cpu    sqlexec
119     beae88   810f04   2     cond wait(await_MC1)      1cpu    scan_1.0
120     a8ab48   8127d8   2     ready                     1cpu    scan_2.0
121     a96850   810460   2     ready                     1cpu    scan_2.1
122     ab6f30   8119a8   2     running                   1cpu    scan_2.2
```

Figure 13-13
onstat -g ath Output

onstat -g act

Use the **onstat -g act** option to obtain a list of active threads. The **onstat -g act** output shows a subset of threads that is also listed in **onstat -g ath** output.

onstat -g ses

Use the **onstat -a ses** option to monitor the resources allocated for and used by a session; in particular, a session that is running a decision-support query. For example, in Figure 13-14, session number 49 is running five threads for a decision-support query.

```
session                                          #RSAM    total      used
id       user     tty      pid   hostname threads  memory     memory
57       informix -        0     -        0        8192       5908
56       user_3   ttyp3    2318  host_10  1        65536      62404
55       user_3   ttyp3    2316  host_10  1        65536      62416
54       user_3   ttyp3    2320  host_10  1        65536      62416
53       user_3   ttyp3    2317  host_10  1        65536      62416
52       user_3   ttyp3    2319  host_10  1        65536      62416
51       user_3   ttyp3    2321  host_10  1        65536      62416
49       user_1   ttyp2    2308  host_10  5        188416     178936
2        informix -        0     -        0        8192       6780
1        informix -        0     -        0        8192       4796
```

Figure 13-14
onstat -g ses Output

onstat -g mem and onstat -g stm

Use the **onstat -g mem** and **onstat -g stm** options to obtain information on the memory used for each session. You can determine which session to focus on by the **used memory** column of the **onstat -g ses** output.

Figure 13-15 shows sample **onstat -g ses** output and an excerpt of the **onstat -g mem** and **onstat -g stm** outputs for Session 16.

- Option **onstat -g mem** shows the total amount of memory used by each session.

  The **totalsize** column of the **onstat -g mem 16** output shows the total amount of memory allocated to the session.

- Option **onstat -g stm** shows the portion of the total memory allocated to the current prepared SQL statement.

  The **heapsz** column of the **onstat -g stm 16** output in Figure 13-15 shows the amount of memory allocated for the current prepared SQL statement.

```
onstat -g ses

session                                        #RSAM    total      used
id       user     tty      pid      hostname threads  memory     memory
18       informix -        0        -        0        12288      8928
17       informix 12       28826    lyceum   1        45056      33752
16       virginia 6        28743    lyceum   1        90112      79504
14       virginia 7        28734    lyceum   1        45056      33096
3        informix -        0        -        0        12288      10168
2        informix -        0        -        0        12288      8928



onstat -g mem 16

Pool Summary:
name         class addr     totalsize freesize #allocfrag #freefrag
16           V     a9ea020  90112     10608    159        5
...


onstat -g stm 16

session   16 --------------------------------------------------------------
 sdblock  heapsz  statement ('*' = Open cursor)
 aa0d018   10056 *SELECT C.customer_num, O.order_num
FROM customer C, orders O, items I
WHERE C.customer_num = O.customer_num
AND O.order_num = I.order_num
```

Figure 13-15
onstat Outputs to Determine Session Memory

UNIX

Using ON-Monitor to Monitor Sessions

Choose **User** from the Status menu. This option provides a subset of the information that the **onstat -u** utility displays. The following information appears:

- The session ID
- The user ID
- The number of locks that the thread is holding
- The number of read calls and write calls that the thread has executed
- Flags that indicate the present state of the thread (for example, waiting for a buffer or waiting for a checkpoint), whether the thread is the primary thread for a session, and what type of thread it is (for example, user thread, daemon thread, and so on)

For sample output, see Figure 13-16.

```
                        USER THREAD INFORMATION

                    Locks       Disk       Disk       User thread
Session   User      Held        Reads      Writes     Status

      0   informix     0          96          2       ------D
      0   informix     0           0          0       ------F
      0   informix     0           0          0       -------
     15   informix     0           0          0       Y-----M
      0   informix     0           0          0       ------D
     17   chrisw       1           3         34       Y------
```

Figure 13-16
Output from the User Option of the ON-Monitor Status Menu

Using ISA to Monitor Sessions

To monitor user sessions with ISA, navigate to the **Users** page and click **Threads**. ISA uses information generated by **onstat -u** display information. Click **Refresh** to rerun the commands and display fresh information.

Using SMI Tables

Query the **syssessions** table to obtain the following information.

Column	Description
sid	Session ID
username	User name (login ID) of the user
uid	User ID
pid	Process ID
connected	Time that the session started
feprogram	Application that is running as the client (front-end program)

In addition, some columns contain flags that indicate if the *primary* thread of the session is waiting for a latch, lock, log buffer, or transaction; if it is an ON-Monitor thread; and if it is in a critical section.

Important: *The information in the* ***syssessions*** *table is organized by session, and the information in the* ***onstat -u*** *output is organized by thread. Also, unlike the* ***onstat -u*** *output, the* ***syssessions*** *table does not include information on daemon threads, only user threads.*

Query the **syssesprof** table to obtain a profile of the activity of a session. This table contains a row for each session with columns that store statistics on session activity (for example, number of locks held, number of row writes, number of commits, number of deletes, and so on).

For a complete list of the **syssessions** columns and descriptions of **syssesprof** columns, see the chapter on the **sysmaster** database in the *Administrator's Reference*.

Monitoring Transactions

Monitor transactions to track open transactions and the locks that those transactions hold. You can use ISA to monitor transactions and user sessions. ISA uses information that the following **onstat** command-line options generate to display session information, as the following table shows. Click the **Refresh** button to rerun the **onstat** command and display fresh information.

To Monitor	Select on ISA	Displays the Output of	Refer to
Transaction statistics	**Users→Transaction**	**onstat -x**	"Displaying Transactions with onstat -x" on page 13-59
User session statistics	**Users→Threads**	**onstat -u**	"Displaying User Sessions with onstat -u" on page 13-62
Lock statistics	**Performance→Locks**	**onstat -k**	"Displaying Locks with onstat -k" on page 13-61
Sessions running SQL statements	**Users→Connections→session-id**	**onstat -g sql** *session-id*	"Displaying Sessions Executing SQL Statements" on page 13-63

Displaying Transactions with onstat -x

The **onstat -x** output contains the following information for each open transaction:

- The address of the transaction structure in shared memory
- Flags that indicate the following information:
  - The present state of the transaction (user thread attached, suspended, waiting for a rollback)
  - The mode in which the transaction is running (loosely coupled or tight coupled)
  - The stage that the transaction is in (BEGIN WORK, prepared to commit, committing or committed, rolling back)
  - The nature of the transaction (global transaction, coordinator, subordinate, both coordinator and subordinate)
- The thread that owns the transaction
- The number of locks that the transaction holds
- The logical-log file in which the BEGIN WORK record was logged
- The current logical-log id and position
- The isolation level
- The number of attempts to start a recovery thread
- The coordinator for the transaction (if the subordinate is executing the transaction)
- The maximum number of concurrent transactions since you last initialized the database server

This utility is especially useful for monitoring global transactions. For example, you can determine whether a transaction is executing in loosely coupled or tightly coupled mode. These transaction modes have the following characteristics:

- Loosely coupled mode

  Each branch in a global transaction has a separate transaction ID (XID). This mode is the default.

  - The different database servers coordinate transactions, but do not share resources. No two transaction branches, even if they access the same database, can share locks.
  - The records from all branches of a global transaction display as separate transactions in the logical log.

- Tightly coupled mode

  In a single global transaction, all branches that access the same database share the same transaction ID (XID). This mode only occurs with the Microsoft Transaction Server (MTS) transaction manager.

  - The different database servers coordinate transactions and share resources such as locks and log records. The branches with the same XID share locks and can never wait on another branch with the same XID because only one branch is active at one time.
  - Log records for branches with the same XID appear under the same transaction in the logical log.

Figure 13-17 shows sample output from **onstat -x**. The last transaction listed is a global transaction, as the G value in the fifth position of the **flags** column indicates. The T value in the second position of the **flags** column indicates that the transaction is running in tightly coupled mode.

```
Transactions
address flags userthread locks beginlg curlog logposit isol    retrys coord
ca0a018 A---- c9da018    0     0       5      0x18484c COMMIT  0
ca0a1e4 A---- c9da614    0     0       0      0x0      COMMIT  0
ca0a3b0 A---- c9dac10    0     0       0      0x0      COMMIT  0
ca0a57c A---- c9db20c    0     0       0      0x0      COMMIT  0
ca0a748 A---- c9db808    0     0       0      0x0      COMMIT  0
ca0a914 A---- c9dbe04    0     0       0      0x0      COMMIT  0
ca0aae0 A---- c9dcff8    1     0       0      0x0      COMMIT  0
ca0acac A---- c9dc9fc    1     0       0      0x0      COMMIT  0
ca0ae78 A---- c9dc400    1     0       0      0x0      COMMIT  0
ca0b044 AT--G c9dc9fc    0     0       0      0x0      COMMIT  0
10 active, 128 total, 10 maximum concurrent
```

Figure 13-17
onstat -x Output

The output in Figure 13-17 shows that this transaction branch is holding 13 locks. When a transaction runs in tightly coupled mode, the branches of this transaction share locks.

Displaying Locks with onstat -k

Use the **onstat -k** option to obtain more details on the locks that a transaction holds. To find the relevant locks, match the address in the **userthread** column in **onstat -x** to the address in the **owner** column of **onstat -k**. Figure 13-18 shows sample output from **onstat -x** and the corresponding **onstat -k** command. The a335898 value in the **userthread** column in **onstat -x** matches the value in the **owner** column of the two lines of **onstat -k**.

```
onstat -x

Transactions
address  flags userthread locks   beginlg curlog  logposit   isol    retrys
coord
a366018  A---- a334018    0      0       1       0x22b048  COMMIT  0
a3661f8  A---- a334638    0      0       0       0x0       COMMIT  0
a3663d8  A---- a334c58    0      0       0       0x0       COMMIT  0
a3665b8  A---- a335278    0      0       0       0x0       COMMIT  0
a366798  A---- a335898    2      0       0       0x0       COMMIT  0
a366d38  A---- a336af8    0      0       0       0x0       COMMIT  0
 6 active, 128 total, 9 maximum concurrent

onstat -k

Locks
address  wtlist   owner    lklist   type     tblsnum  rowid    key#/bsiz
a09185c  0        a335898  0        HDR+S    100002   20a         0
a0918b0  0        a335898  a09185c  HDR+S    100002   204         0
 2 active, 2000 total, 2048 hash buckets, 0 lock table overflows
```

Figure 13-18
onstat -k and onstat -x Output

In the example in Figure 13-18, a user is selecting a row from two tables. The user holds the following locks:

- A shared lock on one database
- A shared lock on another database

Displaying User Sessions with onstat -u

You can find the session-id of the transaction by matching the address in the **userthread** column of the **onstat -x** output with the **address** column in the **onstat -u** output. The **sessid** column of the same line in the **onstat -u** output provides the session id. For example, Figure 13-19 shows the address a335898 in the **userthread** column of the **onstat -x** output. The output line in **onstat -u** with the same address shows the session id 15 in the **sessid** column.

```
onstat -x

Transactions
address  flags userthread locks   beginlg curlog  logposit   isol    retrys
coord
a366018  A---- a334018    0      0       1       0x22b048  COMMIT  0
a3661f8  A---- a334638    0      0       0       0x0       COMMIT  0
a3663d8  A---- a334c58    0      0       0       0x0       COMMIT  0
a3665b8  A---- a335278    0      0       0       0x0       COMMIT  0
a366798  A---- a335898    2      0       0       0x0       COMMIT  0
a366d38  A---- a336af8    0      0       0       0x0       COMMIT  0
 6 active, 128 total, 9 maximum concurrent


onstat -u

address  flags    sessid  user     tty     wait     tout locks nreads
  nwrites
a334018  ---P--D 1        informix -        0        0    0     20       6
a334638  ---P--F 0        informix -        0        0    0     0        1
a334c58  ---P--- 5        informix -        0        0    0     0        0
a335278  ---P--B 6        informix -        0        0    0     0        0
a335898  Y--P---  15      informix 1        a843d70  0    2     64       0
a336af8  ---P--D 11       informix -        0        0    0     0        0
 6 active, 128 total, 17 maximum concurrent
```

Figure 13-19
Obtaining session-id for userthread in onstat -x

For a transaction executing in loosely coupled mode, the first position of the **flags** column in the **onstat -u** output might display a value of T. This T value indicates that one branch within a global transaction is waiting for another branch to complete. This situation could occur if two different branches in a global transaction, both using the same database, tried to work on the same global transaction simultaneously.

For a transaction executing in tightly coupled mode, this T value does not occur because the database server shares one transaction structure for all branches that access the same database in the global transaction. Only one branch is attached and active at one time and does not wait for locks because the transaction owns all the locks held by the different branches.

Displaying Sessions Executing SQL Statements

To obtain information about the last SQL statement that each session executed, issue the **onstat -g sql** command with the appropriate session ID. Figure 13-20 shows sample output for this option using the same session ID obtained from the **onstat -u** sample in Figure 13-19.

```
onstat -g sql 15

Sess  SQL            Current            Iso Lock       SQL  ISAM F.E.
Id    Stmt type      Database           Lvl Mode       ERR  ERR  Vers
15    SELECT         vjp_stores         CR  Not Wait   0    0    9.03

Current statement name : slctcur

Current SQL statement :
  select l.customer_num, l.lname, l.company,   l.phone, r.call_dtime,
    r.call_descr from customer l, vjp_stores@gilroy:cust_calls r where
    l.customer_num = r.customer_num

Last parsed SQL statement :
  select l.customer_num, l.lname, l.company,   l.phone, r.call_dtime,
    r.call_descr from customer l, vjp_stores@gilroy:cust_calls r where
    l.customer_num = r.customer_num
```

Figure 13-20
onstat -g sql Output

Chapter

14

The onperf Utility on UNIX

In This Chapter 14-3

Overview of the onperf Utility 14-3
- Basic onperf Functions 14-3
  - Displaying Metric Values 14-4
  - Saving Metric Values to a File 14-4
  - Reviewing Metric Measurements 14-5
- The onperf Tools 14-6

Requirements for Running onperf. 14-7

Starting and Exiting onperf 14-8

The onperf User Interface 14-9
- Graph Tool 14-9
  - Title Bar 14-10
  - Graph-Tool Graph Menu 14-11
  - Graph-Tool Metrics Menu 14-12
  - Graph-Tool View Menu 14-13
  - The Graph-Tool Configure Menu and the Configuration Dialog Box 14-14
  - Graph-Tool Tools Menu 14-16
  - Changing the Scale of Metrics 14-17
  - Displaying Recent-History Values 14-17
- Query-Tree Tool 14-19
- Status Tool 14-20
- Activity Tools 14-21

Ways to Use onperf 14-21
Routine Monitoring 14-21
Diagnosing Sudden Performance Loss 14-22
Diagnosing Performance Degradation 14-22

The onperf Metrics 14-22
Database Server Metrics 14-23
Disk-Chunk Metrics 14-25
Disk-Spindle Metrics 14-26
Physical-Processor Metrics 14-26
Virtual-Processor Metrics 14-27
Session Metrics 14-27
Tblspace Metrics 14-29
Fragment Metrics 14-30

UNIX

In This Chapter

This chapter describes the **onperf** utility, a windowing environment that you can use to monitor the database server performance. The **onperf** utility monitors the database server running on the UNIX operating system.

The **onperf** utility allows you to monitor most of the same database server metrics that the **onstat** utility reports. The **onperf** utility provides these main advantages over **onstat**:

- Displays metric values graphically in real time
- Allows you to choose which metrics to monitor
- Allows you to scroll back to previous metric values to analyze a trend
- Allows you to save performance data to a file for review at a later time

Overview of the onperf Utility

This section provides an overview of **onperf** functionality and the different **onperf** tools.

Basic onperf Functions

The **onperf** utility performs the following basic functions:

- Displays the values of the database server metrics in a tool window
- Saves the database server metric values to a file
- Allows review of the database server metric values from a file

Displaying Metric Values

When **onperf** starts, it activates the following processes:

- **The onperf process.** This process controls the display of **onperf** tools.
- **The data-collector process.** This process attaches to shared memory and passes performance information to the **onperf** process for display in an **onperf** tool.

An **onperf** tool is a Motif window that an **onperf** process manages, as Figure 14-1 shows.

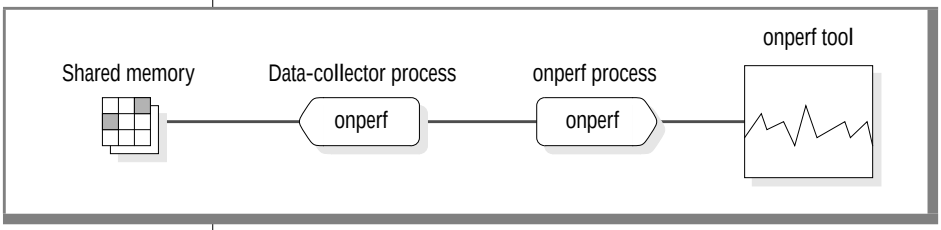


Figure 14-1
Data Flow from Shared Memory to an onperf Tool Window

Saving Metric Values to a File

The **onperf** utility allows designated metrics to be continually buffered. The data collector writes these metrics to a circular buffer called the *data-collector buffer*. When the buffer becomes full, the oldest values are overwritten as the data collector continues to add data. The current contents of the data-collector buffer are saved to a history file, as Figure 14-2 illustrates.

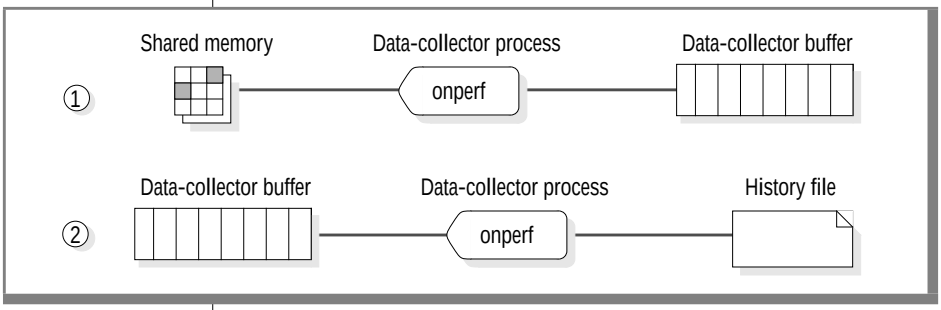


Figure 14-2
How onperf Saves Performance Data

The **onperf** utility uses either a binary format or an ASCII representation for data in the history file. The binary format is host dependent and allows data to be written quickly. The ASCII format is portable across platforms.

You have control over the set of metrics stored in the data-collector buffer and the number of samples. You could buffer all metrics; however, this action might consume more memory than is feasible. A single metric measurement requires 8 bytes of memory. For example, if the sampling frequency is one sample per second, then to buffer 200 metrics for 3,600 samples requires approximately 5.5 megabytes of memory. If this process represents too much memory, you must reduce the depth of the data-collector buffer, the sampling frequency, or the number of buffered metrics.

To configure the buffer depth or the sampling frequency, you can use the Configuration dialog box. For more information on the Configuration dialog box, refer to the "The Graph-Tool Configure Menu and the Configuration Dialog Box" on page 14-14.

Reviewing Metric Measurements

You can review the contents of a history file in a tool window. When you request a tool to display a history file, the **onperf** process starts a playback process that reads the data from disk and sends the data to the tool for display. The playback process is similar to the data-collector process mentioned under "Saving Metric Values to a File" on page 14-4. However, instead of reading data from shared memory, the playback process reads measurements from a history file. Figure 14-3 shows the playback process.

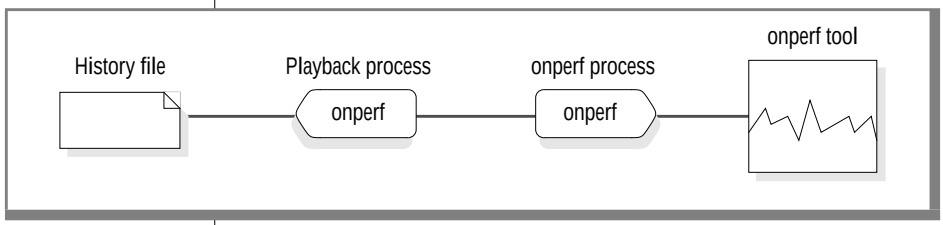


Figure 14-3 *Flow of Data from a History File to an onperf Tool Window*

The onperf Tools

The **onperf** utility provides the following Motif windows, called *tools*, to display metric values:

- Graph tool

  This tool allows you to monitor general performance activity. You can use this tool to display any combination of metrics that **onperf** supports and to display the contents of a history file. For more information, see "Graph Tool" on page 14-9.

- Query-tree tool

  This tool displays the progress of individual queries. For more information, see "Query-Tree Tool" on page 14-19.

- Status tool

  This tool displays status information about the database server and allows you to save the data that is currently held in the data-collector buffer to a file. For more information, see "Status Tool" on page 14-20.

- Activity tools

  These tools display specific database server activities. Activity tools include disk, session, disk-capacity, physical-processor, and virtual-processor tools. The physical-processor and virtual-processor tools, respectively, display information about all CPUs and VPs. The other activity tools each display the top 10 instances of a resource ranked by a suitable activity measurement. For more information, see "Activity Tools" on page 14-21.

Requirements for Running onperf

When you install the database server, the following executable files are written to the **$INFORMIXDIR/bin** directory:

- **onperf**
- **onedcu**
- **onedpu**
- **xtree**

In addition, the onperf.hlp online help file is placed in the directory **$INFORMIXDIR/hhelp**.

When the database server is installed and running in online mode, you can bring up **onperf** tools either on the computer that is running the database server or on a remote computer or terminal that can communicate with your database server instance. Figure 14-4 illustrates both possibilities. In either case, the computer that is running the **onperf** tools must support the X terminal and the **mwm** window manager.

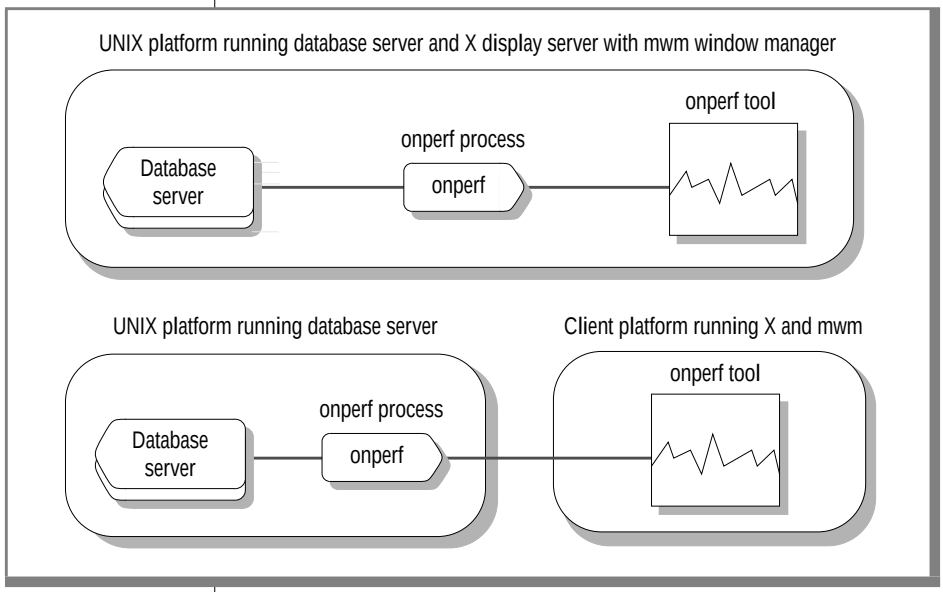


Figure 14-4
Two Options for Running onperf

Starting and Exiting onperf

Before you start **onperf**, set the following environment variables to the appropriate values:

- **DISPLAY**
- **LD_LIBRARY_PATH**

Set the **DISPLAY** environment variable as follows:

```
C shell         setenv DISPLAY displayname0:0 #

Bourne shell    DISPLAY=displayname0:0 #
```

In these commands, *displayname* is the name of the computer or X terminal where the **onperf** window should appear.

Set the **LD_LIBRARY_PATH** environment variable to the appropriate value for the Motif libraries on the computer that is running **onperf**.

With the environment properly set up, you can enter **onperf** to bring up a graph-tool window, as described in "The onperf User Interface" on page 14-9.

To exit **onperf**, use the **Close** option to close each tool window, use the **Exit** option of a tool, or choose **Window Manager→Close**.

You can monitor multiple database server instances from the same Motif client by invoking **onperf** for each database server, as the following example shows:

```
INFORMIXSERVER=instance1 ; export INFORMIXSERVER; onperf
INFORMIXSERVER=instance2 ; export INFORMIXSERVER; onperf
...
```

The onperf User Interface

When you invoke **onperf**, the utility displays an initial graph-tool window. From this graph-tool window, you can display additional graph-tool windows as well as the query-tree, data-collector, and activity tools. The graph-tool windows have no hierarchy; you can create and close these windows in any order.

Graph Tool

The graph tool is the principal **onperf** interface. This tool allows you to display any set of database server metrics that the **onperf** data collector obtains from shared memory. Figure 14-5 shows a diagram of a graph tool.

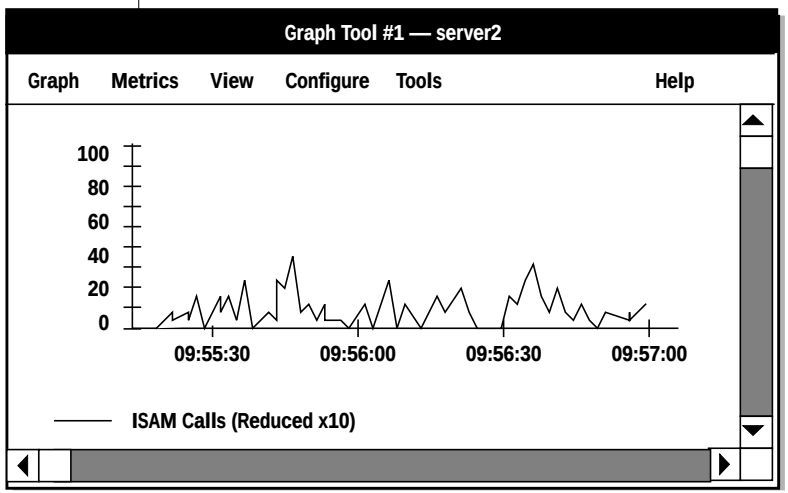


Figure 14-5
Graph-Tool Window

You cannot bring up a graph-tool window from a query-tree tool, a status tool, or one of the activity tools.

Title Bar

All graph-tool windows contain information in the title bar. Figure 14-6 shows the format.

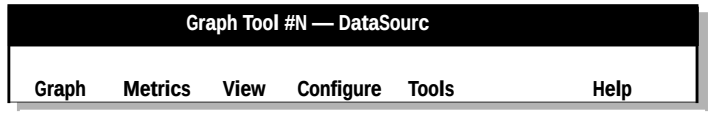


Figure 14-6 *Title-Bar Information*

When you invoke **onperf**, the initial graph-tool window displays a title bar such as the one that Figure 14-7 shows. In this case, ***serverName*** is the database server that the **INFORMIXSERVER** environment variable specifies.

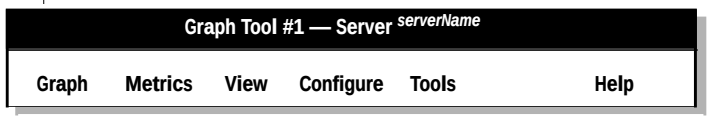


Figure 14-7 *Title Bar for Initial Graph-Tool Window*

Because the configuration of this initial graph-tool has not yet been saved or loaded from disk, **onperf** does not display the name of a configuration file in the title bar.

The data source that Figure 14-7 displays is the database server that the **INFORMIXSERVER** environment variable specifies, meaning that the data comes from the shared memory of the indicated database server instance.

Suppose you open a historical data file named **caselog.23April.2PM** in this graph-tool window. The title bar now displays the information that Figure 14-8 shows.

Figure 14-8 *Title Bar for a Graph-Tool Window That Displays Data from a History File*

Graph-Tool Graph Menu

The **Graph** menu provides the following options.

Option	Use
New	Creates a new graph tool. All graph tools that you create using this option share the same data-collector and **onperf** processes. To create new graph tools, use this option rather than invoke **onperf** multiple times.
Open History File	Loads a previously saved file of historical data into the graph tool for viewing. If the file does not exist, **onperf** prompts you for one. When you select a file, **onperf** starts a playback process to view the file.
Save History File	Saves the contents of the data-collector buffer to either an ASCII or a binary file, as specified in the Configuration dialog box.
Save History File As	Specifies the filename in which to save the contents of the data-collector buffer.
Annotate	Brings up a dialog box in which you can enter a header label and a footer label. Each label is optional. The labels are displayed on the graph. When you save the graph configuration, **onperf** includes these labels in the saved configuration file.
Print	Brings up a dialog box that allows you to select a destination file. You cannot send the contents of the graph tool directly to a printer; you must use this option to specify a file and subsequently send the PostScript file to a printer.
Close	Closes the tool. When a tool is the last remaining tool of the **onperf** session, this menu item behaves in the same way as the **Exit** option.
Exit	Exits **onperf**.

Important: *To save your current configuration before you load a new configuration from a file, you must choose* ***Configure→Save Configuration*** *or* ***Configure→Save Configuration As****.*

Graph-Tool Metrics Menu

Use the **Metrics** menu to choose the class of metrics to display in the graph tool.

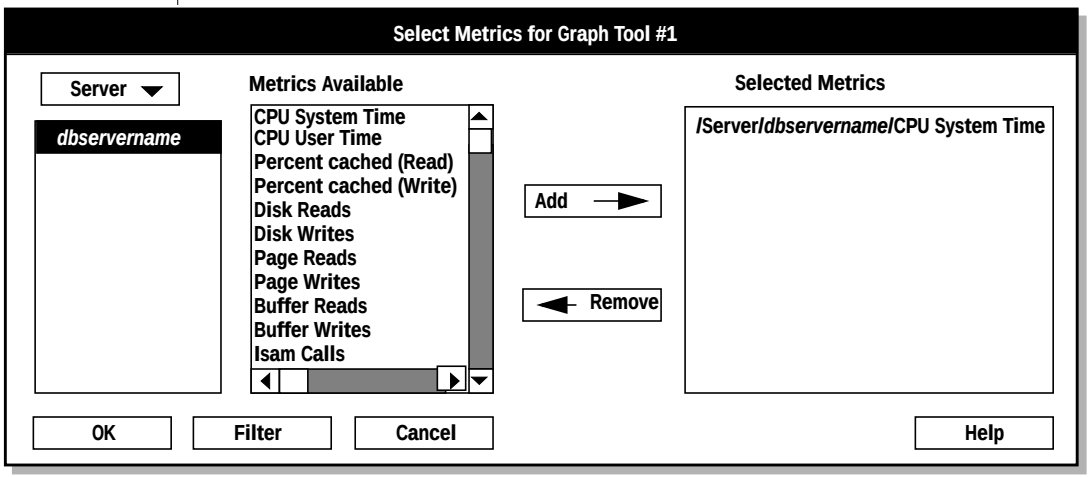
Metrics are organized by *class* and *scope*. When you select a metric for the graph tool to display, you must specify the metric class, the metric scope, and the name of the metric.

The *metric class* is the generic database server component or activity that the metric monitors. The *metric scope* depends on the metric class. In some cases, the metric scope indicates a particular component or activity. In other cases, the scope indicates all activities of a given type across an instance of the database server.

The **Metrics** menu has a separate option for each class of metrics. For more information on metrics, see "Ways to Use onperf" on page 14-21.

When you choose a class, such as **Server**, you see a dialog box like the one in Figure 14-9.

Figure 14-9
The Select Metrics Dialog Box



The Select Metrics dialog box contains three list boxes. The list box on the left displays the valid scope levels for the selected metrics class. For example, when the scope is set to **Server**, the list box displays the dbservername of the database server instance that is being monitored. When you select a scope from this list, **onperf** displays the individual metrics that are available within that scope in the middle list box. You can select one or more individual metrics from this list and add them to the display.

Tip: *You can display metrics from more than one class in a single graph-tool window. For example, you might first select* ***ISAM Calls****,* ***Opens****, and* ***Starts*** *from the* ***Server*** *class. When you choose the* ***Option*** *menu in the same dialog box, you can select another metric class without exiting the dialog box. For example, you might select the Chunks metric class and add the* ***Operations****,* ***Reads****, and* ***Writes*** *metrics to the display.*

The **Filter** button in the dialog box brings up an additional dialog box in which you can filter long text strings shown in the Metrics dialog box. The Filter dialog box also lets you select tables or fragments for which metrics are not currently displayed.

Graph-Tool View Menu

The **View** menu provides the following options.

Option	Use
Line	Changes the graph tool to the line format. Line format includes horizontal and vertical scroll bars. The vertical scroll bar adjusts the scale of the horizontal time axis. When you raise this bar, **onperf** reduces the scale and vice versa. The horizontal scroll bar allows you to adjust your view along the horizontal time axis. To change the color and width of the lines in the line format, click the legend in the graph tool. When you do, **onperf** generates a Customize Metric dialog box that provides a choice of line color and width.
Horizontal Bar Graph	Changes the graph tool to the horizontal bar format.
Vertical Bar Graph	Changes the graph tool to the vertical bar format.

(1 of 2)

Option	Use
Pie	Changes the graph tool to the pie-chart format. To display a pie chart, you must select at least two metrics.
Quick Rescale Axis	Rescales the axis to the largest point that is currently visible on the graph. This button turns off automatic rescaling.
Configure Axis	Displays the Axis Configuration dialog box. Use this dialog box to select a fixed value for the y-axis on the graph or select automatic axis scaling.

(2 of 2)

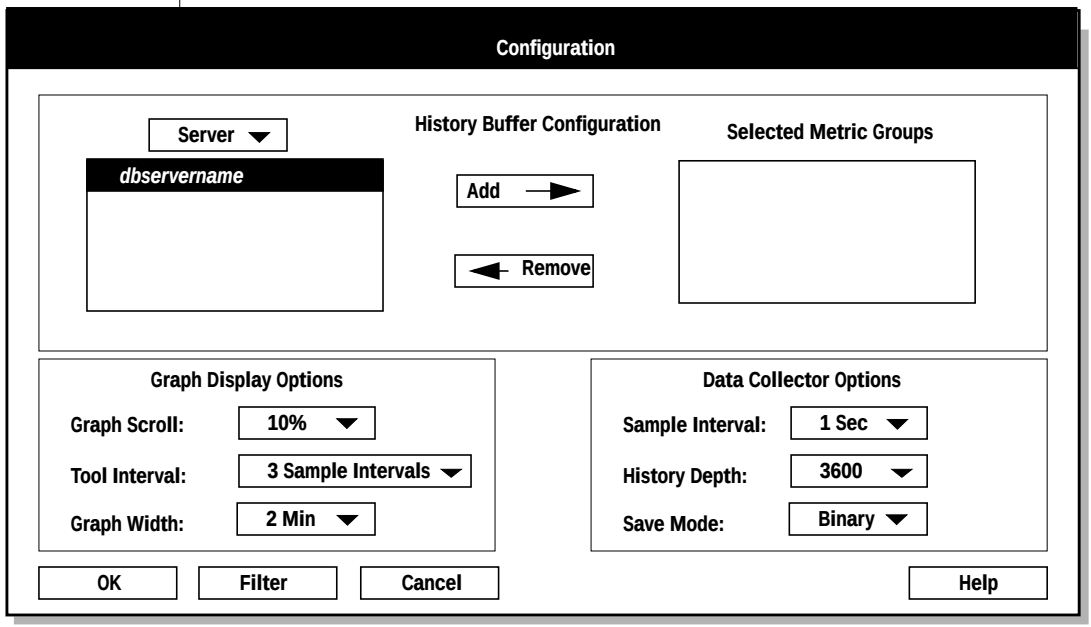
The Graph-Tool Configure Menu and the Configuration Dialog Box

The **Configure** menu provides the following options.

Option	Use
Edit Configuration	Brings up the Configuration dialog box, which allows you to change the settings for the current data-collector buffer, graph-tool display options, and data-collector options. The Configuration dialog box appears in Figure 14-9 on page 14-12.
Open Configuration	Reinitializes **onperf** with the settings that are stored in the configuration file. Unsaved data in the data-collector buffer is lost. If no configuration file is specified and the default does not exist, the following error message appears: `Open file filename failed.` If the specified configuration file does not exist, **onperf** prompts for one.
Save Configuration	Saves the current configuration to a file. If no configuration file is currently specified, **onperf** prompts for one.
Save Configuration As	Saves a configuration file. This option always prompts for a filename.

To configure data-collector options, graph-display options, and metrics about which to collect data, choose the **Edit Configuration** option to bring up the Configuration dialog box.

Figure 14-10
The Configuration Dialog Box



The Configuration dialog box provides the following options for configuring display.

Option	Use
History Buffer Configuration	Allows you to select a metric class and metric scope to include in the data-collector buffer. The data collector gathers information about all metrics that belong to the indicated class and scope.
Graph Display Options	Allows you to adjust the size of the graph portion that scrolls off to the left when the display reaches the right edge, the initial time interval that the graph is to span, and the frequency with which the display is updated.
Data Collector Options	Controls the collection of data. The sample interval indicates the amount of time to wait between recorded samples. The history depth indicates the number of samples to retain in the data-collector buffer. The save mode indicates the data-collector data should be saved in binary or ASCII format.

Graph-Tool Tools Menu

Use the **Tools** menu to bring up other tools. This menu provides the following options.

Option	Use
Query Tree	Starts a query-tree tool. For more information, see "Query-Tree Tool" on page 14-19.
Status	Starts a status tool. For more information, see "Status Tool" on page 14-20.
Disk Activity	Starts a disk-activity tool. For more information, see "Activity Tools" on page 14-21.
Session Activity	Starts a session-activity tool. For more information, see "Activity Tools" on page 14-21.

(1 of 2)

Option	Use
Disk Capacity	Starts a disk-capacity tool. For more information, see "Activity Tools" on page 14-21.
Physical Processor Activity	Starts a physical-processor tool. For more information, see "Activity Tools" on page 14-21.
Virtual Processor Activity	Starts a virtual-processor tool. For more information, see "Activity Tools" on page 14-21.

(2 of 2)

Changing the Scale of Metrics

When **onperf** displays metrics, it automatically adjusts the scale of the y-axis to accommodate the largest value. You can use the Customize Metric dialog box to establish one for the current display. For more information, refer to "Graph-Tool View Menu" on page 14-13.

Displaying Recent-History Values

The **onperf** utility allows you to scroll back over previous metric values that are displayed in a line graph. This feature allows you to analyze a recent trend. The time interval to which you can scroll back is the *lesser* of the following intervals:

- The time interval over which the metric has been displayed
- The history interval that the graph-tool Configuration dialog box specifies

  For more information, see "The Graph-Tool Configure Menu and the Configuration Dialog Box" on page 14-14.

Figure 14-11 illustrates the maximum scrollable intervals for metrics that span different time periods.

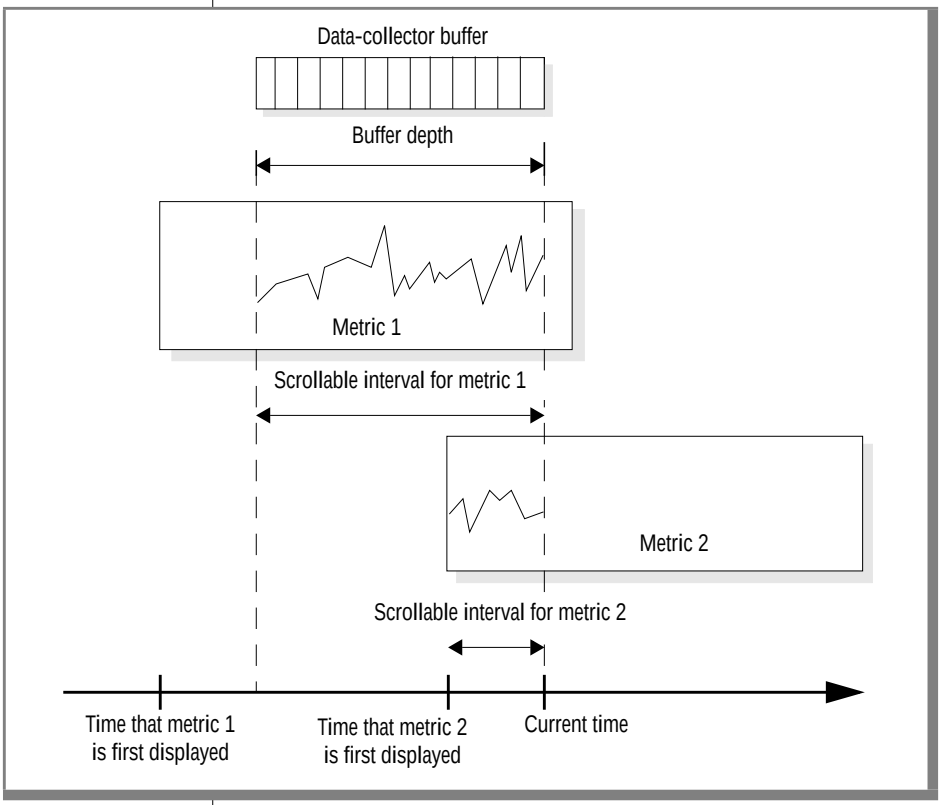


Figure 14-11 *Maximum Scrollable Intervals for Metrics That Span Different Time Periods*

Query-Tree Tool

The query-tree tool allows you to monitor the performance of individual queries. It is a separate executable tool that does not use the data-collector process. You cannot save query-tree tool data to a file. Figure 14-12 shows a diagram of the query-tree tool.

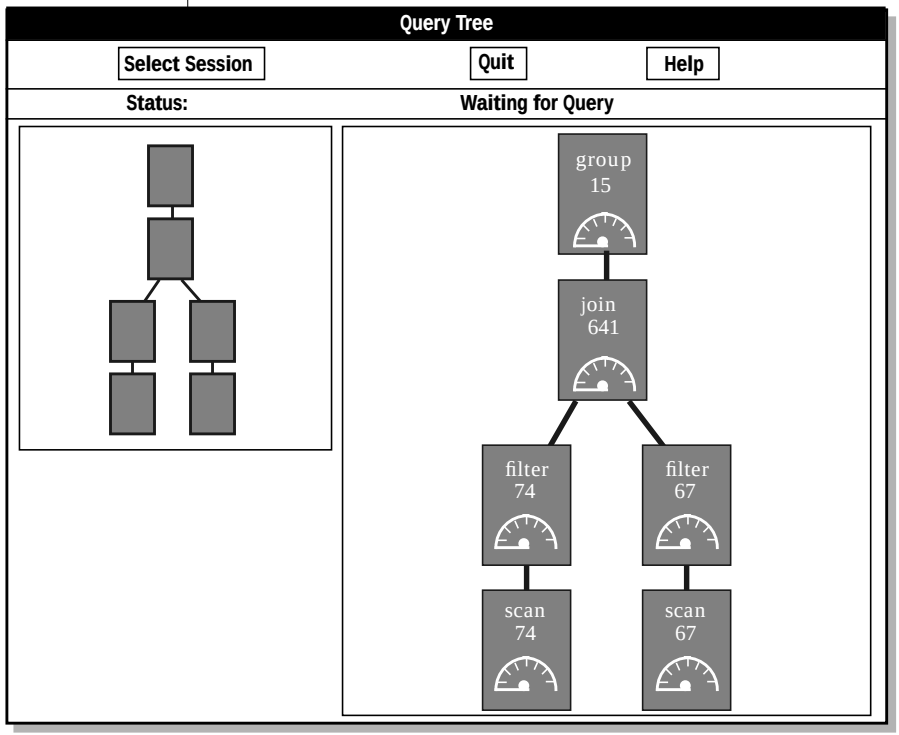


Figure 14-12
Query-Tree Tool Window

This tool includes a **Select Session** button and a **Quit** button. When you select a session that is running a query, the large detail window displays the SQL operators that constitute the execution plan for the query. The query-tree tool represents each SQL operator with a box. Each box includes a dial that indicates rows per second and a number that indicates input rows. In some cases, not all the SQL operators can be represented in the detail window. The smaller window shows the SQL operators as small icons.

The **Quit** button allows you to exit from the query-tree tool.

Status Tool

The status tool allows you to select metrics to store in the data-collector buffer. In addition, you can use this tool to save the data currently held in the data-collector buffer to a file. Figure 14-13 shows a status tool.

The status tool displays:

- The length of time that the data collector has been running
- The size of the data-collector process area, called the *collector virtual memory size*

  When you select different metrics to store in the data-collector buffer, you see different values for the collector virtual memory size.

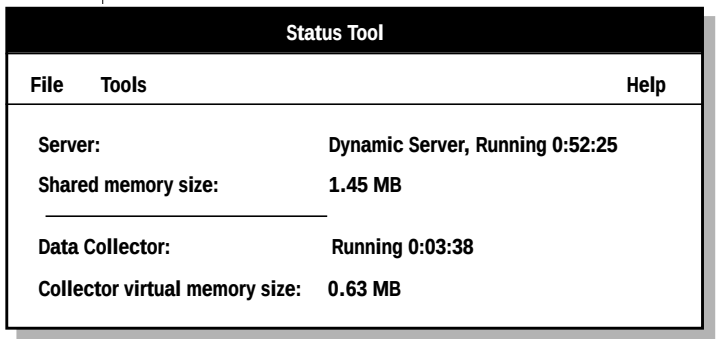


Figure 14-13
Status Tool Window

The status tool **File** menu provides the following options.

Option	Use
Close	This option closes the tool. When it is the last remaining tool of the **onperf** session, Close behaves in the same way as Exit.
Exit	This option exits **onperf**.

Activity Tools

Activity tools are specialized forms of the graph tool that display instances of the specific activity, based on a ranking of the activity by some suitable metric. You can choose from among the following activity tools:

- The disk-activity tool, which displays the top 10 activities ranked by total operations
- The session-activity tool, which displays the top 10 activities ranked by ISAM calls plus PDQ calls per second
- The disk-capacity tool, which displays the top 10 activities ranked by free space in megabytes
- The physical-processor-activity tool, which displays all processors ranked by nonidle CPU time
- The virtual-processor-activity tool, which displays all VPs ranked by VP user time plus VP system time

The activity tools use the bar-graph format. You cannot change the scale of an activity tool manually; **onperf** always sets this value automatically.

The **Graph** menu provides you with options for closing, printing, and exiting the activity tool.

Ways to Use onperf

The following sections describe different ways to use the **onperf** utility.

Routine Monitoring

You can use the **onperf** utility to facilitate routine monitoring. For example, you can display several metrics in a graph-tool window and run this tool throughout the day. Displaying these metrics allows you to monitor database server performance visually at any time.

Diagnosing Sudden Performance Loss

When you detect a sudden performance dip, you can examine the recent history of the database server metrics values to identify any trend. The **onperf** utility allows you to scroll back over a time interval, as explained in "Displaying Recent-History Values" on page 14-17.

Diagnosing Performance Degradation

Performance problems that gradually develop might be difficult to diagnose. For example, if you detect a degradation in database server response time, it might not be obvious from looking at the current metrics which value is responsible for the slowdown. The performance degradation might also be sufficiently gradual that you cannot detect a change by observing the recent history of metric values. To allow for comparisons over longer intervals, **onperf** allows you to save metric values to a file, as explained in "Status Tool" on page 14-20.

The onperf Metrics

The following sections describe the various metric classes. Each section indicates the scope levels available and describes the metrics within each class.

Database server performance depends on many factors, including the operating-system configuration, the database server configuration, and the workload. It is difficult to describe relationships between **onperf** metrics and specific performance characteristics.

The approach taken here is to describe each metric without speculating on what specific performance problems it might indicate. Through experimentation, you can determine which metrics best monitor performance for a specific database server instance.

Database Server Metrics

The scope for these metrics is always the named database server, which means the database server as a whole, rather than a component of the database server or disk space.

Metric Name	Description
CPU System Time	System time, as defined by the platform vendor
CPU User Time	User time, as defined by the platform vendor
Percent Cached (Read)	Percentage of all read operations that are read from the buffer cache without requiring a disk read, calculated as follows: `100 * ((buffer_reads – disk_reads) / (buffer_reads))`
Percent Cached (Write)	Percentage of all write operations that are buffer writes, calculated as follows: `100 * ((buffer_writes – disk_writes) / (buffer_writes))`
Disk Reads	Total number of read operations from disk
Disk Writes	Total number of write operations to disk
Page Reads	Number of pages transferred to disk
Page Writes	Number of pages read from disk
Buffer Reads	Number of reads from the buffer cache
Buffer Writes	Number of writes to the buffer cache
Calls	Number of calls received at the database server
Reads	Number of read calls received at the database server
Writes	Number of write calls received at the database server
Rewrites	Number of rewrite calls received at the database server
Deletes	Number of delete calls received at the database server
Commits	Number of commit calls received at the database server

(1 of 3)

Metric Name	Description
Rollbacks	Number of rollback calls received at the database server
Table Overflows	Number of times that the tblspace table was unavailable (overflowed)
Lock Overflows	Number of times that the lock table was unavailable (overflowed)
User Overflows	Number of times that the user table was unavailable (overflowed)
Checkpoints	Number of checkpoints written since database server shared memory was initialized
Buffer Waits	Number of times that a thread waited to access a buffer
Lock Waits	Number of times that a thread waited for a lock
Lock Requests	Number of times that a lock was requested
Deadlocks	Number of deadlocks detected
Deadlock Timeouts	Number of deadlock timeouts that occurred (Deadlock timeouts involve distributed transactions.)
Checkpoint Waits	Number of checkpoint waits; in other words, the number of times that threads have waited for a checkpoint to complete
Index to Data Pages Read-aheads	Number of read-ahead operations for index keys
Index Leaves Read-aheads	Number of read-ahead operations for index leaf nodes
Data-path-only Read-aheads	Number of read-ahead operations for data pages
Latch Requests	Number of latch requests
Network Reads	Number of ASF messages read
Network Writes	Number of ASF messages written
Memory Allocated	Amount of database server virtual-address space in kilobytes

(2 of 3)

Metric Name	Description
Memory Used	Amount of database server shared memory in kilobytes
Temp Space Used	Amount of shared memory allocated for temporary tables in kilobytes
PDQ Calls	The total number of parallel-processing actions that the database server performed
DSS Memory	Amount of memory currently in use for decision-support queries

(3 of 3)

Disk-Chunk Metrics

The disk-chunk metrics take the pathname of a chunk as the metric scope.

Metric Name	Description
Disk Operations	Total number of I/O operations to or from the indicated chunk
Disk Reads	Total number of reads from the chunk
Disk Writes	Total number of writes to the chunk
Free Space (MB)	The amount of free space available in megabytes

Disk-Spindle Metrics

The disk-spindle metrics take the pathname of a disk device or operation-system file as the metric scope.

Metric Name	Description
Disk Operations	Total number of I/O operations to or from the indicated disk or buffered operating-system file
Disk Reads	Total number of reads from the disk or operating-system file
Disk Writes	Total number of writes to the disk or operating-system file
Free Space	The amount of free space available in megabytes

Physical-Processor Metrics

The physical-processor metrics take either a physical-processor identifier (for example, `0` or `1`) or **Total** as the metric scope.

Metric Name	Description
Percent CPU System Time	CPU system time for the physical processors
Percent CPU User Time	CPU user time for the physical processors
Percent CPU Idle Time	CPU idle time for the physical processors
Percent CPU Time	The sum of CPU system time and CPU user time for the physical processors

Virtual-Processor Metrics

These metrics take a virtual-processor class as a metric scope (cpu, aio, kaio, and so on). Each metric value represents a sum across all instances of this virtual-processor class.

Metric Name	Description
User Time	Accumulated user time for a class
System Time	Accumulated system time for a class
Semaphore Operations	Total count of semaphore operations
Busy Waits	Number of times that virtual processors in class avoided a context switch by spinning in a loop before going to sleep
Spins	Number of times through the loop
I/O Operations	Number of I/O operations per second
I/O Reads	Number of read operations per second
I/O Writes	Number of write operations per second

Session Metrics

These metrics take an active session as the metric scope.

Metric Name	Description
Page Reads	Number of pages read from disk on behalf of a session
Page Writes	Number of pages written to disk on behalf of a session
Number of Threads	Number of threads currently running for the session
Lock Requests	Number of lock requests issued by the session
Lock Waits	Number of lock waits for session threads

(1 of 2)

Metric Name	Description
Deadlocks	Number of deadlocks involving threads that belong to the session
Deadlock timeouts	Number of deadlock timeouts involving threads that belong to the session
Log Records	Number of log records written by the session
ISAM Calls	Number of ISAM calls by session
ISAM Reads	Number of ISAM read calls by session
ISAM Writes	Number of ISAM write calls by session
ISAM Rewrites	Number of ISAM rewrite calls by session
ISAM Deletes	Number of ISAM delete calls by session
ISAM Commits	Number of ISAM commit calls by session
ISAM Rollbacks	Number of ISAM rollback calls by session
Long Transactions	Number of long transactions owned by session
Buffer Reads	Number of buffer reads performed by session
Buffer Writes	Number of buffer writes performed by session
Log Space Used	Amount of logical-log space used
Maximum Log Space Used	High-watermark of logical-log space used for this session
Sequential Scans	Number of sequential scans initiated by session
PDQ Calls	Number of parallel-processing actions performed for queries initiated by the session
Memory Allocated	Memory allocated for the session in kilobytes
Memory Used	Memory used by the session in kilobytes

(2 of 2)

Tblspace Metrics

These metrics take a tblspace name as the metric scope. A tblspace name is composed of the database name, a colon, and the table name (*database:table*). For fragmented tables, the tblspace represents the sum of all fragments in a table. To obtain measurements for an individual fragment in a fragmented table, use the Fragment Metric class.

Metric Name	Description
Lock Requests	Total requests to lock tblspace
Lock Waits	Number of times that threads waited to obtain a lock for the tblspace
Deadlocks	Number of times that a deadlock involved the tblspace
Deadlock Timeouts	Number of times that a deadlock timeout involved the tblspace
Reads	Number of read calls that involve the tblspace
Writes	Number of write calls that involve the tblspace
Rewrites	Number of rewrite calls that involve the tblspace
Deletes	Number of delete calls that involve the tblspace
Calls	Total calls that involve the tblspace
Buffer Reads	Number of buffer reads that involve tblspace data
Buffer Writes	Number of buffer writes that involve tblspace data
Sequential Scans	Number of sequential scans of the tblspace
Percent Free Space	Percent of the tblspace that is free
Pages Allocated	Number of pages allocated to the tblspace
Pages Used	Number of pages allocated to the tblspace that have been written
Data Pages	Number of pages allocated to the tblspace that are used as data pages

Fragment Metrics

These metrics take the dbspace of an individual table fragment as the metric scope.

Metric Name	Description
Lock Requests	Total requests to lock fragment
Lock Waits	Number of times that threads have waited to obtain a lock for the fragment
Deadlocks	Number of times that a deadlock involved the fragment
Deadlock Timeouts	Number of times that a deadlock timeout involved the fragment
Reads	Number of read calls that involve the fragment
Writes	Number of write calls that involve the fragment
Rewrites	Number of rewrite calls that involve the fragment
Deletes	Number of delete calls that involve the fragment
Calls	Total calls that involve the fragment
Buffer Reads	Number of buffer reads that involve fragment data
Buffer Writes	Number of buffer writes that involve fragment data
Sequential Scans	Number of sequential scans of the fragment
Percent Free Space	Percent of the fragment that is free
Pages Allocated	Number of pages allocated to the fragment
Pages Used	Number of pages allocated to the fragment that have been written to
Data Pages	Number of pages allocated to the fragment that are used as data pages

Appendix

A

Case Studies and Examples

This appendix contains a case study and several extended examples of performance-tuning methods that this manual describes.

Case Study

The following case study illustrates a situation in which the disks are overloaded. It shows the steps taken to isolate the symptoms and identify the problem based on an initial report from a user, and it describes the needed correction.

A database application that has not achieved the desired throughput is being examined to see how performance can be improved. The operating-system monitoring tools reveal that a high proportion of process time was spent idle, waiting for I/O. The database server administrator increases the number of CPU VPs to make more processors available to handle concurrent I/O. However, throughput does not increase, which indicates that one or more disks are overloaded.

To verify the I/O bottleneck, the database server administrator must identify the overloaded disks and the dbspaces that reside on those disks.

To identify overloaded disks and the dbspaces that reside on those disks

1. To check the asynchronous I/O (AIO) queues, use **onstat -g ioq**. Figure A-1 shows the output.

```
AIO I/O queues:
q name/id    len maxlen totalops  dskread dskwrite  dskcopy
opt    0       0      0        0        0        0        0
msc    0       0      0        0        0        0        0
aio    0       0      0        0        0        0        0
pio    0       0      1        1        0        1        0
lio    0       0      1      341        0      341        0
gfd    3       0      1      225        2      223        0
gfd    4       0      1      225        2      223        0
gfd    5       0      1      225        2      223        0
gfd    6       0      1      225        2      223        0
gfd    7       0      0        0        0        0        0
gfd    8       0      0        0        0        0        0
gfd    9       0      0        0        0        0        0
gfd   10       0      0        0        0        0        0
gfd   11       0     28    32693    29603     3090        0
gfd   12       0     18    32557    29373     3184        0
gfd   13       0     22    20446    18496     1950        0
```

Figure A-1
Output from the onstat -g ioq Option

In Figure A-1, the **maxlen** and **totalops** columns show significant results:

- The **maxlen** column shows the largest backlog of I/O requests to accumulate within the queue. The last three queues are much longer than any other queue in this column listing.
- The **totalops** column shows 100 times more I/O operations completed through the last three queues than for any other queue in the column listing.

The **maxlen** and **totalops** columns indicate that the I/O load is severely unbalanced.

Another way to check I/O activity is to use **onstat -g iov**. This option provides a slightly less detailed display for all VPs.

2. To check the AIO activity for each disk device, use **onstat -g iof**, as Figure A-2 shows.

```
AIO global files:
gfd pathname          totalops  dskread dskwriteio/s
3 /dev/infx2               0        0        00.0
4 /dev/infx2               0        0        00.0
5 /dev/infx2               2        2        00.0
6 /dev/infx2             223        0      2230.5
7 /dev/infx4               0        0        00.0
8 /dev/infx4               1        0        10.0
9 /dev/infx4             341        0      3410.7
10 /dev/infx4              0        0        00.0
11 /dev/infx5          32692    29602     309067.1
12 /dev/infx6          32556    29372     318466.9
13 /dev/infx7          20446    18496     195042.0
```

Figure A-2
Output from the onstat -g iof Option

This output indicates the disk device associated with each queue. Depending on how your chunks are arranged, several queues can be associated with the same device. In this case, the total activity for **/dev/infx2** is the sum of the **totalops** column for queues 3, 4, 5, and 6, which is 225 operations. This activity is still insignificant when compared with **/dev/infx5**, **/dev/infx6**, and **/dev/infx7**.

3. To determine the dbspaces that account for the I/O load, use **onstat -d,** as Figure A-3 shows.

```
Dbspaces
address  number   flags    fchunk   nchunks  flags    owner    name
c009ad00 1        1        1        1        N        informix rootdbs
c009ad44 2        2001     2        1        N T      informix tmp1dbs
c009ad88 3        1        3        1        N        informix oltpdbs
c009adcc 4        1        4        1        N        informix histdbs
c009ae10 5        2001     5        1        N T      informix tmp2dbs
c009ae54 6        1        6        1        N        informix physdbs
c009ae98 7        1        7        1        N        informix logidbs
c009aedc 8        1        8        1        N        informix runsdbs
c009af20 9        1        9        3        N        informix acctdbs
 9 active, 32 total

Chunks
address  chk/dbs offset   size     free     bpages   flags pathname
c0099574 1   1   500000   10000    9100               PO-   /dev/infx2
c009960c 2   2   510000   10000    9947               PO-   /dev/infx2
c00996a4 3   3   520000   10000    9472               PO-   /dev/infx2
c009973c 4   4   530000   250000   242492             PO-   /dev/infx2
c00997d4 5   5   500000   10000    9947               PO-   /dev/infx4
c009986c 6   6   510000   10000    2792               PO-   /dev/infx4
c0099904 7   7   520000   25000    11992              PO-   /dev/infx4
c009999c 8   8   545000   10000    9536               PO-   /dev/infx4
c0099a34 9   9   250000   450000   4947               PO-   /dev/infx5
c0099acc 10  9   250000   450000   4997               PO-   /dev/infx6
c0099b64 11  9   250000   450000   169997             PO-   /dev/infx7
 11 active, 32 total
```

Figure A-3
Display from the onstat -d option

In the **Chunks** output, the **pathname** column indicates the disk device. The **chk/dbs** column indicates the numbers of the chunk and dbspace that reside on each disk. In this case, only one chunk is defined on each of the overloaded disks. Each chunk is associated with dbspace number 9.

The **Dbspaces** output shows the name of the dbspace that is associated with each dbspace number. In this case, all three of the overloaded disks are part of the **acctdbs** dbspace.

Although the original disk configuration allocated three entire disks to the **acctdbs** dbspace, the activity within this dbspace suggests that three disks are not enough. Because the load is about equal across the three disks, it does not appear that the tables are necessarily laid out badly or improperly fragmented. However, you could get better performance by adding fragments on other disks to one or more large tables in this dbspace or by moving some tables to other disks with lighter loads.

Index

Numerics

64-bit addressing
  BUFFERS 4-15
  tuning RESIDENT
    parameter 4-23

A

Access methods
  ANSI-compliant name 7-26
  list of 7-23
  secondary 7-23 to 7-27
Access plan
  description of 10-4
  different types of 10-14
  directives 11-8
  effects of OPTCOMPIND 10-26
  in SET EXPLAIN output 12-30
  SET EXPLAIN output 10-17
  subquery 10-19
Activity tool (onperf)
  description of 14-6
  using 14-21
Administrator, database server. *See* Database server administrator.
ADTERR configuration
    parameter 5-60
ADTMODE configuration
    parameter 5-60
Affinity
  setting for processor 3-13
  VPCLASS configuration
    parameter 3-12
AFF_NPROCS configuration
    parameter 3-9
AFF_SPROC configuration
    parameter 3-9
AIO queues A-2
AIO virtual processors (VPs)
  monitoring 3-30
AIO VPs 3-14, 3-15
Algorithm, in-place alter 6-44, 10-33
ALTER FRAGMENT statement
  eliminating index build during
    DETACH 9-37 to 9-38
  least-cost index build during
    ATTACH 9-31, 9-32, 9-34, 9-35, 9-36
  moving table 6-6
  releasing space 6-45
ALTER FUNCTION statement
  parallel UDRs 13-39
ALTER INDEX statement 6-43, 6-45, 7-15
  compared to CREATE INDEX 6-44
  TO CLUSTER clause 6-43
ALTER TABLE statement
  adding or dropping a column 6-44
  changing data type 6-44
  changing extent sizes 6-35, 6-37
  changing lock mode 8-7, 8-8
  changing sbspace characteristics 6-33
  columns part of an index 6-59
  fast alter algorithm 6-61
  in-place alter algorithm 6-44, 6-52, 10-33

PUT clause 6-33
sbspace fragmentation 9-11
smart large objects 9-11
ANSI compliance
level Intro-17
ANSI Repeatable Read isolation level 8-11
ANSI Serializable isolation level 8-11
ANSI-compliant database, access-method name 7-26
Application developer
concurrency 1-25
encountering locks 8-9, 8-12
forcing use of index 8-12
general responsibility 1-25
isolation level 8-9, 8-12
phantom row 8-10
setting PDQ priority 12-15
SQLWARN array 5-41
Assigning table to a dbspace 6-6
Attached index
creating 9-17
description of 9-17
extent size 7-4
fragmentation 9-18
physical characteristics 9-18
Auditing
and performance 5-60
facility 1-8
AUDITPATH configuration parameter 5-60
AUDITSIZE configuration parameter 5-60

B

Background I/O activities, description of 5-41
Backup and restore
and table placement 6-11, 9-11
fragmentation for 9-8
BAR_IDLE_TIMEOUT parameter 5-57
BAR_MAX_BACKUP parameter 5-57
BAR_NB_XPORT_COUNT parameter 5-57
BAR_PROGRESS_FREQ parameter 5-57
BAR_XFER_BUF_SIZE parameter 5-57
Benchmarks, for throughput 1-8
BLOB data type
definition of 5-9
See also Smart large objects.
Blobpage
determining fullness 5-24
fullness explained 5-26
interpreting average fullness 5-26
oncheck -pB display 5-24
size and storage efficiency 5-24
storage statistics 5-24
Blobpages
estimating number in tblspace 6-17
logical log size 5-51
size 5-23
sizing in blobspace 5-22
when to store in blobspace 6-19
Blobspace
advantages over dbspace 5-21
configuration effects 5-20
determining fullness 5-23
disk I/O 6-19
simple large objects 6-18
specifying in CREATE TABLE 5-21
storage statistics 5-24
unbuffered 6-19
when to use 6-19
blob. *See* Simple large objects.
Branch index pages 7-5
B-tree
btree cleaner 13-26
description of 7-5
estimating index pages 7-3 to 7-9
generic 7-24, 13-38
index usage 7-23
Buffer cache rate 4-57
Buffer pool
64-bit addressing 4-15, 4-23
BUFFERS configuration parameter 4-14
bypass with light scans 5-38
bypass with lightweight I/O 5-33
fuzzy pages 5-57
LRU queues 5-56
memory-resident pages 4-56, 4-57
network 3-22, 3-24
read cache rate 4-17
restrictions with simple large objects 6-19
size for smart large objects 5-29
smart large objects 4-16, 5-29, 5-32
Buffered logging 5-10
BUFFERS
64-bit addressing 4-15
Buffers
data-replication 4-56
free network 3-25
lightweight I/O 5-32, 5-33
logical log 4-21, 5-29
memory-resident 4-57
monitoring network 3-25
network 3-24
physical log 4-21
smart large objects and 5-33
TCP/IP connections 3-22
BUFFERS configuration parameter
description 4-14
monitoring 4-17
reducing disk I/O 4-17
Built-in data types
B-tree index 7-10, 13-23 to 13-26
functional index 7-10
generic B-tree index 7-25
BYTE data type
blobspace 5-20
buffer pool restriction 6-19
estimating table size 6-13
locating 6-18
memory cache 5-34
on disk 6-18
parallel access 5-28
staging area 5-35
storing 6-18
See also Simple large objects.
Byte lock 8-15
Byte-range locking
description 8-23
monitoring 8-25
setting 8-24
B+ tree. *See* B-tree.

C

Caches
  aggregate 10-43
  data dictionary 4-29, 4-30, 4-33
  data distribution 4-31, 4-35
  description 4-27
  opclass 10-43
  storage location 4-6
  typename 10-43
  UDR 10-42
Cardinality changes, and UPDATE STATISTICS 13-14
Case studies, extended A-1 to A-4
Central processing unit (CPU)
  configuration parameters that affect 3-8
  connections and 3-32, 3-34
  environment variables that affect 3-8
  utilization and 1-17
  VPs and 3-27
CHAR data type
  converting to VARCHAR 6-62
  GLS recommendations 10-36
  key-only scans 10-4
Checking indexes 7-21
Checkpoint
  full 5-42
  fuzzy 5-42, 5-46
Checkpoints
  configuration parameters affecting 5-42
  effect on physical log 5-46
  logging and performance 5-48
  monitoring 5-43
  specifying interval 5-42
  when occur 5-42, 5-43
Chunks
  and dbspace configuration 5-5
  and disk partitions 5-6
  critical data 5-7
  monitoring 6-41
CKPINTVL configuration parameter 5-42, 5-43
Class name, virtual processors 3-10
CLEANERS configuration parameter 5-56
CLOB data type 5-9
  See also Smart large objects.
Clustering
  description of 7-15
  index for sequential access 10-31
Code set, ISO 8859-1 Intro-5
Code, sample, conventions for Intro-13
Collection 10-19
Collection scan 10-20
Collection-derived tables
  description of 10-19
  query plan for 10-20
Columns
  filter expression, with join 10-9
  filtered 7-13
  with duplicate keys 7-14
Commands, UNIX
  cron 4-12
  iostat 2-6
  ps 2-6
  sar 2-6, 4-17
  time 1-12
  vmstat 2-6, 4-17
Comment icons Intro-12
COMMIT WORK statement 1-8
Committed Read isolation level 5-39
Complex query, example of 10-15
Compliance
  with industry standards Intro-17
Composite index 13-23, 13-24
  order of columns 13-24
  use of 13-23
Concurrency
  definition of 8-3
  effects of isolation level 8-9, 10-5
  fragmentation 9-4
  page lock on index 8-5
  page locks 8-19
  row and key locks 8-4
  table locks 8-6, 8-19
Configuration parameters
  ADTERR 5-60
  ADTMODE 5-60
affecting
  auditing 5-60
  backup and restore 5-57
  checkpoints 5-42
  connections 3-24
  CPU 3-8
  critical data 5-11
  data dictionary 4-30, 4-33
  data distributions 4-33, 4-35
  data replication 5-59
  ipcshm connection 3-6, 3-21, 4-9
  logging I/O 5-48
  logical log 5-12
  memory 4-12
  multiple connections 3-33
  network free buffer 3-24
  ON-Bar 5-57
  page cleaning 5-55
  physical log 5-12
  poll threads 3-4, 3-19, 3-27
  recovery 5-58
  root dbspace 5-11
  sequential I/O 5-40
  SQL statement cache 4-46, 13-44, 13-45
  SQL statement cache cleaning 4-41, 4-46
  SQL statement cache hits 4-28, 4-40, 4-41, 4-42, 4-43, 4-44, 4-45, 4-48, 4-49, 4-50, 4-53, 4-54
  SQL statement cache memory 4-28, 4-41
  SQL statement cache pools 4-50, 4-51
  SQL statement cache size 4-28, 4-41, 4-48
  SQL statement memory limit 4-48
  UDR cache buckets 4-28, 10-42, 10-44
  UDR cache entries 4-28, 10-42, 10-44
AFF_NPROCS 3-9
AFF_SPROC 3-9
and CPU 3-3
AUDITPATH 5-60
AUDITSIZE 5-60
BAR_IDLE_TIMEOUT 5-57

BAR_MAX_BACKUP 5-57
BAR_NB_XPORT_COUNT 5-57
BAR_PROGRESS_FREQ 5-57
BAR_XFER_BUF_SIZE 5-57
BUFFERS 4-14
CKPINTVL 5-42, 5-43
CLEANERS 5-56
controlling PDQ resources 12-10
DATASKIP 5-41
DBSPACETEMP 5-12, 5-15, 5-17, 6-10, 7-19
DD_HASHMAX 4-30
DD_HASHSIZE 4-30, 4-33
DEADLOCK_TIMEOUT 8-21
DEF_TABLE_LOCKMODE 8-7, 8-8
DIRECTIVES 11-18
DRAUTO 5-59
DRINTERVAL 5-59
DRLOSTFOUND 5-59
DRTIMEOUT 5-59
DS_HASHSIZE 4-33, 4-35
DS_MAX_QUERIES 3-17
DS_MAX_SCANS 3-18, 12-10, 12-11, 12-18
DS_POOLSIZE 4-33, 4-35
DS_TOTAL_MEMORY 4-18, 7-19, 12-10
FILLFACTOR 7-9
INFORMIXOPCACHE 5-37
LOCKS 4-5, 4-21, 8-18
LOGBUFF 4-21, 5-12, 5-29, 5-48
LOGFILES 5-43
LOGSIZE 5-49, 5-50
LOGSSIZE 5-43
LRUS 5-56
LRU_MAX_DIRTY 5-56
LRU_MIN_DIRTY 5-56
LTAPEBLK 5-58
LTAPEDEV 5-58
LTAPESIZE 5-58
LTXEHWM 5-54
LTXHWM 5-54
MAX_PDQPRIORITY 3-9, 3-16, 12-14, 12-18, 12-21, 13-30
MIRROR 5-11
MIRROROFFSET 5-11
MIRRORPATH 5-11
MULTIPROCESSOR 3-11
NETTYPE 3-4, 3-6, 3-21, 3-24, 3-27, 3-33, 4-9
NOAGE 3-9
NUMAIOVPS 3-9
NUMCPUVPS 3-9
OFF_RECVRY_THREADS 5-58
ONDBSPDOWN 5-47
ON_RECVRY_THREADS 5-58
OPCACHEMAX 5-35, 5-36
OPTCOMPIND 3-8, 3-16, 11-18, 12-20
OPT_GOAL 13-34
PC_HASHSIZE 4-28, 10-42, 10-44
PC_POOLSIZE 4-28, 10-42, 10-44
PHYSBUFF 4-21, 5-29, 5-48
PHYSFILE 5-44
RA_PAGES 5-40, 5-55
RA_THRESHOLD 5-40, 5-55
RESIDENT 4-23
ROOTNAME 5-11
ROOTOFFSET 5-11
ROOTPATH 5-11
ROOTSIZE 5-11
SBSPACENAME 5-19, 6-31
SBSPACETEMP 5-18, 5-19, 5-20
SHMADD 4-6, 4-24
SHMBASE 4-12
SHMTOTAL 4-4, 4-6, 4-24
SHMVIRTSIZE 4-6, 4-25
SINGLE_CPU_VP 3-11
STACKSIZE 4-26
STAGEBLOB 5-35, 5-36
STMT_CACHE 13-44, 13-45
STMT_CACHE_HITS 4-28, 4-40, 4-41, 4-42, 4-43, 4-44, 4-45, 4-48, 4-49, 4-50, 4-53, 4-54
STMT_CACHE_NOLIMIT 4-28, 4-41, 4-48
STMT_CACHE_NUMPOOL 4-5 0, 4-51
STMT_CACHE_SIZE 4-28, 4-41, 4-46, 4-48
TAPEBLK 5-58
TAPEDEV 5-58
TAPESIZE 5-58
VPCLASS 3-9, 3-10, 3-12, 3-13, 3-15
Configuration, evaluating 2-3
CONNECT statement 5-7, 6-6
Connection type
  ipcshm 3-4, 3-6, 3-21
  specifying 3-19, 3-20
Connections
  and CPU 3-32, 3-34
  improving performance with MaxConnect 3-34
  multiplexed 3-32, 3-33
  specifying number of 3-20, 3-21
Contact information Intro-17
Contention
  cost of reading a page 10-30
  reducing with fragmentation 9-6
Contiguous disk space
  allocation 6-38
Contiguous extents
  allocation 6-36
  performance advantage of 5-30, 6-25, 6-34, 6-41, 6-42
Contiguous space
  eliminating interleaved extents 6-43
Conventions, documentation Intro-11
Correlated subquery, effect of PDQ 12-9
Cost of user-defined routine 13-38, 13-39, 13-40
Cost per transaction 1-14
CPU utilization
  improving with MaxConnect 3-34
CPU VP class and NETTYPE 3-20
CPU VPs
  and fragmentation goals 9-4
  configuration parameters that affect 3-10
  effect on CPU utilization 3-27
  limited by MAX_PDQPRIORITY 3-16
  limited by PDQ priority 3-8
  optimal number of 3-11
  used by PDQ 12-12
CREATE CLUSTERED INDEX statement 7-16

CREATE FUNCTION statement
parallel UDRs 13-39
selectivity and cost 13-40
specifying stack size 4-26
virtual-processor class 3-10
CREATE INDEX statement
attached index 9-17
compared to ALTER INDEX 6-44
detached index 9-18
FILLFACTOR clause 7-9
generic B-tree index 7-24
parallel build 12-7
TO CLUSTER clause 6-43
USING clause 7-27
CREATE PROCEDURE statement
optimizing SPL routines 10-38
CREATE PROCEDURE statement, SQL optimization 10-38
CREATE TABLE statement
blobspace assignment 5-21
creating system catalog table 5-7
extent sizes 6-35
fragmentation 9-17, 9-18
IN DBSPACE clause 6-6
PUT clause 6-33
sbspace characteristics 6-33
sbspace extents 5-31
sbspace fragmentation 9-11
simple large objects 6-18
smart large objects 9-11
TEMP TABLE clause 5-12, 5-19
USING clause 7-37
CREATE TEMP TABLE statement 9-20
Critical data
configuration parameters that affect 5-11
description of 5-47
introduced 5-7
mirroring 5-8
Critical media
mirroring 5-8
separating 5-8
Critical resource 1-15
Critical section of code
logging 5-44
related to size of physical log 5-45
cron UNIX scheduling facility 2-6, 2-7, 4-12
Cursor Stability isolation level 5-39, 8-11

D

Data conversion 10-35
Data dictionary
DD_HASHMAX 4-30
DD_HASHSIZE 4-30
parameters affecting cache for 4-33
Data distributions
creating 10-22
creating on filtered columns 11-6, 13-15
dropping 13-15
effect on memory 4-6
filter selectivity 10-23
guidelines to create 13-16
how optimizer uses 10-22
join columns 13-18
multiple columns 13-22
parameters affect cache for 4-33, 4-35
sbspace 13-19
syscolumns 13-15, 13-19
sysdistrib 13-15
user-defined data type 13-19
user-defined statistics 13-19, 13-40
Data migration between fragments 9-33
Data replication
and performance 5-59
buffers 4-56
Data transfers per second 1-20
Data type
CHAR 10-4
NCHAR 10-32
NVARCHAR 10-32
VARCHAR 10-4
Data types
BLOB 5-9
built-in, distinct, and opaque 7-23
BYTE 5-20, 6-13, 6-18
CHAR 6-62, 10-36
CLOB 5-9
effect of mismatch 10-35
for simple large objects 6-18
NCHAR 6-62
NVARCHAR 6-15
TEXT 5-20, 6-13, 6-18, 6-63
VARCHAR 6-15, 6-62, 10-36
Database server administrator
allocating DSS memory 12-17
controlling DSS resources 3-17, 12-22
creating staging-area blobspace 5-35
halting database server 5-47
limiting DSS resources 12-13
limiting number of DSS queries 12-19
limiting number of scan threads 12-18
limiting PDQ priority 12-18, 12-21
marking dbspace down 5-47
placing system catalog tables 5-7
responsibility of 1-24, 5-5
specifying unavailable fragments 9-7
using MAX_PDQPRIORITY 12-21
DATABASE statement 5-7, 6-6
DataBlade API functions, smart large objects 5-27, 5-31, 6-31, 6-37, 8-28
DataBlade module
functional index 7-29
new index 7-31
secondary access method 7-23
user-defined index 7-11, 7-22
Data-collector
buffer 14-4
process 14-4

Data-dictionary cache
  advantages 4-29
  configuring 4-30, 4-33
  description 4-29
  effect on SHMVIRTSIZE 4-7
  monitoring 4-30
Data-distribution cache
  description 4-31
  effect on SHMVIRTSIZE 4-7
  monitoring 4-35
DATASKIP configuration
    parameter 5-41
DB-Access utility Intro-5, 2-7, 6-42
dbload utility 6-42, 7-17
DB-Monitor utility 5-35
dbschema utility
  data distributions 9-14
  distribution output 13-20, 13-21, 13-22
  examining value distribution 9-9
Dbspaces
  and chunk configuration 5-5
  configuration parameters
      affecting root 5-11
  for temporary tables and sort
      files 5-12, 6-10
  mirroring root 5-9
  multiple disks within 6-9
  reorganizing to prevent extent
      interleaving 6-42
DBSPACETEMP
  advantages over
      PSORT_DBTEMP 5-16
  configuration parameter 5-12,
      5-15, 6-10, 7-19
  environment variable 5-12, 6-10,
      7-19
  overriding configuration
      parameter 5-15
  parallel inserts 5-12, 12-6
DBUPSPACE environment
    variable 13-22
DD_HASHMAX configuration
    parameter 4-30
DD_HASHSIZE configuration
    parameter 4-30, 4-33
Deadlock 8-21
DEADLOCK_TIMEOUT
    configuration parameter 8-21
Decision-support queries
  monitoring threads for 13-54
Decision-support queries (DSS)
  balanced with transaction
      throughput 1-11
  controlling resources 12-22
  effects of
      DS_TOTAL_MEMORY 4-18
  gate information 12-26
  gate numbers 12-27
  monitoring resources
      allocated 12-23, 12-29
  monitoring resources allocated
      for 13-55
  monitoring threads for 12-27,
      12-28
  performance impact 1-14
  use of temporary files 9-9, 9-10
Default locale Intro-5
DEF_TABLE_LOCKMODE
    configuration parameter 8-7,
    8-8
Demonstration databases Intro-5
Denormalizing
  data model 6-61
  tables 6-62
Dependencies, software Intro-4
Detached index
  extent size 7-4
Detached index, description
    of 9-18, 9-19
Dimensional tables, definition
    of 13-24
DIRECTIVES configuration
    parameter 11-18
Directives. *See* Optimizer directives.
Dirty Read isolation level 5-39, 8-15
Disk access
  cost of reading row 10-30
  performance 13-28
  performance effect of 10-30
  sequential 13-28
  sequential forced by query 13-8
Disk extent
  for dbspaces 6-34
  for sbspaces 5-30
Disk I/O
  allocating AIO VPs 3-15
  background database server
      activities 1-5
  balancing 5-14, 5-19
  big buffers, how used for 4-6
  binding AIO VPs 3-14
  blobspace data and 5-21
  buffered in shared memory 6-19
  BUFFERS configuration
      parameter 4-14
  contention 10-30
  effect of UNIX configuration 3-6
  effect of Windows NT
      configuration 3-7
  effect on performance 5-5
  for temporary tables and sort
      files 5-13
  hot spots, definition of 5-5
  in query plan cost 10-4, 10-12,
      10-21
  isolating critical data 5-8
  KAIO 3-15
  light scans 5-38
  lightweight I/O 5-33
  log buffer size, effect of 5-10
  logical log 5-34
  mirroring, effect of 5-8
  monitoring AIO VPs 3-15
  nonsequential access, effect
      of 7-14
  query response time 1-10
  read-ahead configuration
      parameters 5-40
  reducing 4-17, 6-62
  sbspace data and 5-28
  seek time effect 5-37
  sequential scans 5-37
  simple large objects 5-22, 5-23
  smart large objects 5-30, 5-32
  to physical log 5-11
  TPC-A benchmark 1-9
  unbuffered devices 5-16
Disk layout
  and backup 6-11, 9-8
  and table isolation 6-7
Disk space
  storing TEXT and BYTE data 5-24
Disk utilization, discussion of 1-20

Disks
and saturation 5-6
critical data 5-7
middle partitions 6-8
multiple within dbspace 6-9
partitions and chunks 5-6
Distinct data type 7-22
DISTINCT keyword 13-24
Distributed queries
improving performance of 13-26
remote path in SET EXPLAIN 10-14
used with PDQ 12-10
Distribution scheme
description of 9-3
designing 9-12, 9-13, 9-14
methods described 9-11 to 9-12
Documentation notes Intro-15
Documentation notes, program item Intro-16
Documentation, types of Intro-14
documentation notes Intro-15
machine notes Intro-15
release notes Intro-15
DRAUTO configuration parameter 5-59
DRINTERVAL configuration parameter 5-59
DRLOSTFOUND configuration parameter 5-59
DROP INDEX statement, releasing an index 13-29
Dropping indexes 7-17
DRTIMEOUT configuration parameter 5-59
DSS. *See* Decision-support queries (DSS).
DS_HASHSIZE configuration parameter 4-33, 4-35
DS_MAX_QUERIES
and index build performance 7-19
and MGM 12-10
changing value 12-14
configuration parameter 3-17
limit query number with 12-19
DS_MAX_SCANS
and MGM 12-10
changing value 12-14
configuration parameter 3-18, 12-11, 12-18
scan threads 12-11
DS_POOLSIZE configuration parameter 4-33, 4-35
DS_TOTAL_MEMORY
and DS_MAX QUERIES 3-17
and MAX_PDQPRIORITY 12-13
and MGM 12-10
changing value 12-14
configuration parameter 4-18, 7-19
estimating value for 4-18, 12-17
setting for DSS applications 12-22
setting for OLTP 12-17
dtcurrent() ESQL / C function, to get current date and time 1-13
Duplicate index keys, performance effects of 7-14
Dynamic lock allocation 4-5, 4-21, 8-18
Dynamic log file allocation
benefits 5-52
preventing hangs from rollback of long transaction 5-52
size of new log 5-53

E

Environment variables
affecting
CPU 3-8
I / O 5-16
multiplexed connections 3-33
network buffer pool 3-23, 3-24
network buffer size 3-23, 3-26
parallel sorts 5-17
sort files 5-15
sorting 5-7, 5-13
SQL statement cache 13-44, 13-45
temporary tables 5-7, 5-13, 5-15
boldface type Intro-11
DBSPACETEMP 5-7, 5-12, 5-15, 6-10, 7-19
DBUPSPACE 13-22
FET_BUF_SIZE 13-27
IFX_DEF_TABLE_LOCKMODE 8-7, 8-8
IFX_DIRECTIVES 11-18
IFX_SESSION_MUX 3-33
INFORMIXOPCACHE 5-34, 5-37
OPTCOMPIND 3-8, 3-16, 12-20
OPT_GOAL 13-34
PDQPRIORITY
adjusting value 12-14
for UPDATE STATISTICS 13-22
limiting resources 3-8
overriding default 12-13
parallel sorts 13-30
requesting PDQ resources 12-12
setting PDQ priority 7-18
PSORT_DBTEMP 5-16
PSORT_NPROCS 3-9, 5-17, 7-18, 13-30
STMT_CACHE 13-44, 13-45
en_us.8859-1 locale Intro-5
equal() function 7-33
Equality expression, definition of 9-24
ESQL / C functions, for smart large objects 5-27, 5-31, 6-31, 6-37, 8-28
Estimating space
index extent size 7-4
sbspaces 6-20
smart large objects 6-20
EXECUTE PROCEDURE statement 10-41
Explicit temporary table 9-20
Expression-based distribution scheme
and fragment elimination 9-25
definition of 9-11
designing 9-14
type to use 9-21
Extensibility enhancements Intro-8
EXTENT SIZE clause 6-35
Extents
allocating 6-36
attached index 9-19
eliminating interleaved 6-42
index of fragmented table 9-17

interleaved 6-41
managing 6-34
monitoring 6-41
next-extent size 6-35
performance 5-30, 6-34, 6-41
reclaiming empty space 6-44
reorganizing dbspace to prevent interleaving 6-42
size for attached index 7-4
size for detached index 7-4
size limit 6-39
size of 6-35
sizes for fragmented table 9-9
upper limit on number of 6-39

F

Fact tables, in star schema 13-24
Fast recovery, configuration effects 5-58
Feature icons Intro-12
Features of this product, new Intro-6
FET_BUF_SIZE environment variable 13-27
File descriptors 3-6
Files
dbspaces for sort 6-10
executables for onperf 14-7
in /tmp directory 5-13
saving performance metrics 14-4
TEMP or TMP user environment variable 5-13
$INFORMIXDIR/bin 14-7
FILLFACTOR
configuration parameter 7-9
CREATE INDEX 7-9, 13-26
Filter
columns 10-9
columns in large tables 7-13
definition of 10-23
description of 13-7
effect on performance 13-8
effect on sorting 10-29
evaluated from index 13-23
index used to evaluate 10-24
memory used to evaluate 10-28
query plan 11-4
selectivity defined 10-23
selectivity estimates 10-23
user-defined routines 13-7
finderr utility Intro-16
Flattened subquery 10-18
Forced residency 4-23
Foreground write 5-55
Formula
blobpage size 6-17
buffer pool size 4-15
connections per poll thread 3-21
CPU utilization 1-17
data buffer size, estimate of 4-5
decision-support queries 12-17
disk utilization 1-20
DS total memory 4-19, 4-20
extends, upper limit 6-39
file descriptors 3-6
index extent size 7-4
index pages 6-15, 7-8
initial stack size 4-26
LOGSIZE 5-50
memory grant basis 12-18
message portion 4-9
minimum DS memory 4-19
number of remainder pages 6-14
operating-system shared memory 4-10
paging delay 1-19
partial remainder pages 6-15
quantum of memory 4-18, 12-11, 12-23
RA_PAGES 5-40
RA_THRESHOLD 5-40
resident portion 4-5
resources allocated 3-16
rows per page 6-13
scan threads 12-12
scan threads per query 3-18, 12-18
semaphores 3-5
service time 1-16
shared-memory estimate 12-17
shared-memory increment size 4-24
size of physical log 5-44
sort operation, costs of 10-28
threshold for free network buffers 3-24
Fragment
elimination
definition of 9-21
equality expressions 9-24
fragmentation expressions 9-22
range expressions 9-23
ID
and index entry 7-6
definition of 9-19
for fragmented table 9-10
space estimates 9-10
sysfragments information 9-19
nonoverlapping
on multiple columns 9-28
on single column 9-26
overlapping, on single column 9-27
FRAGMENT BY EXPRESSION clause 9-17
Fragmentation
FRAGMENT BY EXPRESSION clause 9-17, 9-18
goals of 9-4
improving ATTACH operation 9-29 to 9-36
improving DETACH operation 9-37 to 9-38
index restrictions 9-20
indexes, attached 9-17
indexes, detached 9-18
monitoring I/O requests 9-39
monitoring with onstat 9-39
next-extent size 9-16
no data migration during ATTACH 9-33
of smart large objects 9-11
reducing contention 9-6
setting priority levels for PDQ 12-20
sysfragments system catalog 9-39
table name when monitoring 9-39
TEMP TABLE clause 9-20
temporary tables 9-20
See also Fragmentation strategy.

Fragmentation strategy
ALTER FRAGMENT ATTACH clause 9-30 to 9-36
ALTER FRAGMENT DETACH clause 9-37 to 9-38
distribution schemes for fragment elimination 9-21
for finer granularity of backup and restore 9-8
for increased availability of data 9-7
for indexes 9-17
for temporary tables 9-20
how data used 9-8
improving 9-15
planning 9-3
See also Fragmentation.
Freeing shared memory 4-11
Full checkpoint, description of 5-42
Functional index
creating 7-28 to 7-30
DataBlade module 7-29
user-defined function 7-10
using 7-28, 13-7
Function, ESQL/C, dtcurrent() 1-13
Fuzzy checkpoint
description 5-42
effect on physical log 5-46
Fuzzy checkpoints
and fast recovery 5-58
LRU_MAX_DIRTY and LRU_MIN_DIRTY 5-56
Fuzzy pages, monitoring 5-57

G

Generic B-tree index
extending 7-25
parallel UDRs 13-38
user-defined data 7-10
when to use 7-24
Global file descriptor (gfd) queues 3-30
Global Language Support (GLS) Intro-5
Graph tool (onperf)
bar graph 14-13
description of 14-6, 14-9
metric
changing line color and width 14-13
changing scale 14-17
class and scope 14-12
pie chart 14-14
greaterthan() function 7-33
greaterthanorequal() function 7-33
GROUP BY clause
composite index used for 13-23
indexes for 10-25, 13-30
MGM memory 12-11

H

Hash join
in directives 11-5, 11-9
plan example 10-5
temporary space 5-16
when used 10-6
Help Intro-14
High-Performance Loader (HPL) 6-6
History, recent performance 14-17
Home pages, in indexes 6-14
Host variables
SQL statement cache 13-42
Hot spots, description of 5-5

I

Icons
feature Intro-12
Important Intro-12
platform Intro-12
product Intro-12
Tip Intro-12
Warning Intro-12
IFX_DEF_TABLE_LOCKMODE environment variable 8-7, 8-8
IFX_DIRECTIVES environment variable 11-18
IFX_NETBUF_PVTPOOL_SIZE environment variable 3-23, 3-24
IFX_NETBUF_SIZE environment variable 3-23, 3-26
IFX_SESSION_MUX environment variable 3-33
Important paragraphs, icon for Intro-12
IN DBSPACE clause 6-6
Indexes
adding for performance 7-13
and filtered columns 7-13
and snowflake or star schemas 13-25
attached index extent size 7-4
autoindex
for inner table 10-5
path 10-14
replacing with permanent 13-23
avoiding duplicate keys 7-14
B-tree cleaner to balance nodes 13-26
checking 7-21
choosing columns for 7-13
composite 13-23, 13-24
cost of on NCHAR 10-32
cost of on NVARCHAR 10-32
cost of on VARCHAR 10-4
DataBlade module 7-31
detached index extent size 7-4
disk space used by 7-11, 13-29
distinct types 7-22
dropping 6-50, 7-17
duplicate entries 7-14
effect of physical order of table rows 10-11
effect of updating 13-26
estimating pages 7-6
estimating space 7-3 to 7-9
extent size 7-4
functional 7-28, 13-7
impact on delete, insert, and update operations 7-12
key-only scan 10-4
managing 7-11
memory-resident 4-57
on CHAR column 10-4
on fact table in star schema 13-25
opaque data types 7-22
order-by and group-by columns 7-14

ordering columns in composite 13-24
placement on disk 7-3
size estimate 7-6
structure of entries 7-6
time cost of 7-12
user-defined data types 7-22 to 7-37
when not used by optimizer 10-35, 13-8
when replaced by join plans 10-10
when to rebuild 13-26
Industry standards, compliance with Intro-17
Informix MaxConnect (IMC)
description 3-34
Informix Server Administrator ISA)
monitoring latches 4-59
Informix Server Administrator (ISA)
capabilities 2-8
creating blobspaces 5-21
creating staging-area blobspace 5-35
description 2-8
generating UPDATE STATISTICS statements 13-13
monitoring I/O Utilization 2-15
monitoring optical cache 5-35
monitoring SQL statement cache 4-42
monitoring user sessions 2-19, 13-58
monitoring virtual processors 3-31
starting virtual processors 3-27
$INFORMIXDIR/hhelp directory 14-7
$INFORMIXDIR/bin directory 14-7
INFORMIXDIR/bin directory Intro-6
INFORMIXOPCACHE environment variable 5-34, 5-37
Inner table
directives for 11-10, 11-11
index for 10-5
In-place alter algorithm
performance advantages 6-52
when used 6-53, 6-54, 6-56
Input 28
Input-output (I/O)
background activities 5-41
contention and high-use tables 6-7
disk saturation 5-6
tables, configuring 5-37
Input/output (I/O)
See also Disk I/O.
Input/output (I/O). *See* Disk I/O.
INSERT cursor 9-11
Interleaved extents 6-41
INTO TEMP clause of the SELECT statement 5-13, 5-14, 5-19
iostat command 2-6
ipcshm connection 3-21, 4-9
ISO 8859-1 code set Intro-5
Isolating tables 6-7
Isolation level
monitoring 2-19
Isolation levels
ANSI Repeatable Read 8-11
ANSI Serializable 8-11
Committed Read 5-39, 8-10
Cursor Stability 5-39, 8-11
Dirty Read 5-39, 8-10, 8-15
effect on concurrency 10-5
effect on joins 10-5
light scans 5-39
monitoring 8-22
Repeatable Read 5-39, 8-11
Repeatable Read and OPTCOMPIND 10-26, 12-20
SET ISOLATION statement 8-9

J

Join
avoiding 13-8
column for composite index 13-24
directives 11-8
effect of large join on optimization 13-36
hash join 10-5
hash join, when used 10-6
in subquery 12-16
in view 12-16
nested-loop join 10-5
order 10-7, 11-4, 11-7, 11-15
outer 12-16
parallel execution 12-15
running UPDATE STATISTICS on columns 13-18
semi join 10-19
SET EXPLAIN output 12-19
subquery flattening 10-18
thread for 12-4
three-way 10-7
with column filters 10-9
See also Join plan.
Join and sort, reducing impact of 13-29
Join method 10-4, 10-26
Join plan
definition of 10-4
different types of 10-14
directive precedence 11-18
directives 11-10
effects of OPTCOMPIND 10-26
hash 10-14, 11-14, 11-15, 12-20, 12-30
hash, in directives 11-5, 11-9
isolation level effect 10-5
nested-loop 10-14, 11-9, 11-12, 11-14
OPTCOMPIND 12-20
optimizer choosing 11-4
replacing index use 10-7
selected by optimizer 10-3
star 13-25
subquery 10-19
See also Join.

K

Kernel asynchronous I/O (KAIO) 3-15
Key-first scan 10-17
Key-only index scan 10-4, 10-17, 10-46

L

Latches
  description 4-58
  monitoring 4-58, 4-59
Latency, disk I/O 10-30
Leaf index pages, description of 7-5
Least-recently-used (LRU)
  and memory-resident tables 4-56
  memory-management algorithm 1-18
  monitoring 5-56
  queues 5-56
  thresholds for I/O to physical log 5-11
lessthan() function 7-33
lessthanorequal() function 7-33
Light scans
  advantages 5-38
  description 5-38
  isolation level 5-39
  mentioned 4-13
  when occur 5-39
Lightweight I/O
  specifying in onspaces 5-33
  specifying with LO_NOBUFFER flag 5-33
  when to use 4-16, 5-32, 5-33, 5-34, 5-54
LIKE test 13-8
LOAD and UNLOAD statements 6-6, 6-42, 6-45, 7-17
Locale Intro-5
  default Intro-5
  en_us.8859-1 Intro-5
Locating simple large objects 6-18
Lock table
  specifying initial size 4-5, 4-21
Locks
  blobpage 5-22
  byte 8-15
  byte-range 8-23
  changing lock mode 8-7
  concurrency 8-3
  database 8-7
  determining owner 8-20
  duration 8-9
  dynamic allocation 4-5, 4-21, 8-18
  effects of table lock 8-6
  exclusive 8-6, 8-15
  granularity 8-3
  initial number 8-18
  intent 8-16
  internal lock table 8-10, 8-15
  isolation level 8-9
  isolation levels and join 10-5
  key-value 8-4
  longspins 2-12
  monitoring by session 8-19
  not waiting for 8-9
  page 8-5
  promotable 8-13
  retaining update locks 8-14
  row and key 8-4
  shared 8-6, 8-15
  specifying mode 8-7, 8-8
  table 8-6
  types 8-15
  update 8-15
  waiting for 8-9
LOCKS configuration parameter 4-5, 4-21, 8-18
LOGBUFF configuration parameter 4-21, 5-12, 5-29, 5-48
LOGFILES configuration parameter
  effect on checkpoints 5-43
  use in logical-log size determination 5-49
Logging
  checkpoints 5-48
  configuration effects 5-48
  critical section of code 5-44
  dbspaces 5-50
  I/O activity 5-29
  LOGSIZE configuration parameter 5-49, 5-50
  none with SBSPACETEMP configuration parameter 5-18, 5-19
  simple large objects 5-21, 5-51
  smart large objects 5-51
  with SBSPACENAME configuration parameter 5-19
Logical log
  and data-replication buffers 4-56
  assigning files to a dbspace 5-8
  buffer size 4-21
  buffered 5-10
  configuration parameters that affect 5-12
  determining disk space allocated 5-49
  logging mode 5-10
  mirroring 5-10
  simple large objects 5-51
  smart large objects 5-51
  unbuffered 5-10
  viewing records 1-8
LOGSSIZE configuration parameter 5-43
Long transactions
  ALTER TABLE operation 6-53
  configuration effects on 5-52, 5-54
  dynamic log effects on 5-54
  log thresholds 5-54
  LTXHWM configuration parameter 6-51
  preventing hangs from rollback of 5-52
Loosely-coupled mode 13-60
LO_NOBUFFER flag, specifying lightweight I/O 5-33
LO_TEMP flag
  temporary smart large object 5-18
LRU queues and buffer table 4-17
LRUS configuration parameter 5-56
LRU_MAX_DIRTY configuration parameter 5-56
LRU_MIN_DIRTY configuration parameter 5-56
LTAPEBLK configuration parameter 5-58

LTAPEDEV configuration parameter 5-58
LTAPESIZE configuration parameter 5-58
LTXEHWM configuration parameter 5-54
LTXHWM configuration parameter 5-54

M

Machine notes Intro-15
Managing extents 6-34
Materialized view
  description 10-33
  involving table hierarchy 10-46
MAX_PDQPRIORITY
  and MGM 12-10
  and PDQPRIORITY 3-9, 12-15, 12-21
  changing value 12-14
  configuration parameter 3-16
  effects on transaction throughput 3-17
  for DSS query limits 12-13, 12-14
  increasing OLTP resources 12-15
  limiting concurrent scans 12-18
  limiting PDQ resources 5-17, 13-30
  limiting user-requested resources 12-21
Memory
  activity costs 10-28
  aggregate cache 10-43
  and data-replication buffers 4-56
  caches 4-27
  configuration parameters that affect 4-12
  data-dictionary cache 4-7, 4-29, 4-30
  data-distribution cache 4-7
  estimate for sorting 7-19
  for SPL routines 12-17
  hash join 5-16
  increase by logging 5-34
  limited by MAX_PDQPRIORITY 3-16
  limited by PDQ priority 3-8
  limited by STMT_CACHE_NOLIMIT 4-41
  limited by STMT_CACHE_SIZE 4-41
  monitoring by session 12-29
  monitoring MGM allocation 12-11
  network buffer pool 3-22, 3-24, 3-26
  opclass cache 10-43
  PDQ priority effect on 7-20, 12-13
  private network free-buffer pool 3-23, 3-24
  quantum allocated by MGM 12-11, 12-13, 12-22, 12-23, 12-25
  sort 4-7
  SQL statement cache 4-7, 13-41
  SSC limit 4-41
  SSC size 4-41
  typename 10-43
  UDR cache 4-7, 10-42
  UNIX configuration parameters 3-6
  utilization 1-18
  Windows NT parameters 3-7
Memory Grant Manager (MGM)
  description of 12-10
  DSS queries 12-11
  memory allocated 4-18
  monitoring resources 12-11, 12-23
  scan threads 12-11
  sort memory 7-20
Memory-management system 1-18
Memory-resident indexes, description of 4-56
Memory-resident pages, buffer pool 4-56
Memory-resident tables
  description of 4-56
  monitoring 4-57
Message
  portion of shared memory 4-9
  queues 4-6
Message file for error messages Intro-16
Metadata
  improving I/O for smart large objects 6-23
Metadata area in sbspace
  contents of 6-20
  estimating size 6-21, 6-22, 6-23
  logging 5-51
  mirroring 5-9
  reserved space for 6-21
Metric classes, onperf
  database server 14-23
  disk chunk 14-25
  disk spindle 14-26
  fragment 14-30
  physical processor 14-26
  session 14-27
  tblspace 14-29
  virtual processor 14-27
Microsoft Transaction Server
  global transactions Intro-7
  tightly coupled mode 13-60
MIRROR configuration parameter 5-11, 5-13
Mirroring
  and critical media 5-8
  root dbspace 5-9, 5-13
  sbspace 5-9
MIRROROFFSET configuration parameter 5-11
MIRRORPATH configuration parameter 5-11
MODIFY NEXT SIZE clause 6-35, 6-37
Monitoring
  aggregate cache 10-43
  AIO virtual processors 3-30
  buffer pool 4-17
  BUFFERS 4-17
  checkpoints 5-43
  CPU utilization 2-12
  data-dictionary cache 4-30
  data-distribution cache 4-35
  deadlocks 8-21
  disk utilization 2-15
  estimated number of rows 10-13
  foreground writes 5-55
  fragments 9-39
  fuzzy pages 5-57
  global transactions 13-60, 13-62

isolation level 2-19
I/O queues for AIO VPs 3-15
latch waits 4-58, 4-59
light scans 5-39
lock owner 8-20
locks 2-19, 8-17, 8-18, 8-20, 13-58
locks used by sessions 8-19
logical-log files 2-18
longspins 2-12
LRU queues 5-56
memory per thread 4-8
memory pools 4-7
memory usage 4-8
memory utilization 2-13
memory-resident tables 4-57
MGM resources 12-23
network buffer size 3-26
network buffers 3-25
number of connections 5-50
number of users 5-44
OPCACHEMAX 5-35
optical cache 5-35
PDQ threads 12-27, 12-28
resources for a session 12-29
sbspace metadata size 6-22
sbspaces 6-25, 6-29
session memory 2-20, 4-8, 4-54, 4-55, 13-46, 13-47, 13-48, 13-49, 13-52, 13-55
sessions 13-52, 13-55
smart large objects 6-25
sort files 2-15
SPL routine cache 10-43, 10-44
SQL statement cache 4-42, 4-43, 4-45, 4-53, 13-51
SQL statement cache entries 13-50
SQL statement cache pool 4-50, 4-51, 4-52
SQL statement cache size 4-46, 4-51
SSC memory pools 4-42
STAGEBLOB blobspace 5-35
statement memory 2-20, 13-46, 13-49
temporary dbspaces 2-15
threads
  concurrent users 4-8
  per CPU VP 3-11
  session 3-11, 12-27, 13-52, 13-54
throughput 1-8
transaction 2-19
transactions 2-19, 13-58
UDR cache 10-43, 10-44
user sessions 2-19
user threads 2-19, 13-52, 13-53, 13-58, 13-59
virtual portion 4-9
virtual processors 3-27, 3-28, 3-29, 3-30, 3-31
Monitoring database server
  active tblspaces 6-38
  blobspace storage 5-24
  buffers 4-17
  sessions 2-20, 13-52
  threads 2-11, 13-52
  transactions 13-58
  virtual processors 3-27
Monitoring the statement cache 4-44
Monitoring tools
  database server utilities 2-6, 2-7
  UNIX 2-6
  Windows NT 2-6
Motif window manager 14-4, 14-6, 14-8
Multiple residency, avoiding 3-3
Multiplexed connection
  description of 3-32
  how to use 3-33
  performance improvement with 3-33
MULTIPROCESSOR configuration parameter 3-11
Mutex, waiting threads 2-12
mwm window manager, required for onperf 14-7

N

NCHAR data type 6-62
Negator function 13-41
Nested-loop join 10-5, 11-9
NET VP class and NETTYPE 3-20
NETTYPE configuration parameter
  connections 3-24, 4-26
  estimating LOGSIZE 5-50
  ipcshm connection 3-6, 3-21, 4-9
  multiple connections 3-33
  network free buffer 3-24
  poll threads 3-4, 3-27
  specifying connections 3-19 to 3-22
Network
  as performance bottleneck 2-4
  buffer size 3-26
  common buffer pool 3-22, 3-26
  communication delays 5-5
  connections 3-6, 3-19
  free-buffer threshold 3-24, 3-25
  monitoring buffers 3-25
  multiplexed connections 3-32
  performance issues 10-36
  private free-buffer pool 3-23, 3-24
Network buffer pools 3-22, 3-24
New features of this product Intro-6
NEXT SIZE clause 6-35
NOAGE configuration parameter 3-9
NOFILE, NOFILES, NFILE, or NFILES operating-system configuration parameters 3-6
NUMAIOVPS configuration parameter 3-9
NUMCPUVPS configuration parameter 3-9
NVARCHAR data type 6-15
  in table-size estimates 6-15

O

obtaining 4-5, 4-15, 5-45, 6-13
OFF_RECVRY_THREADS
  configuration parameter 5-58
OLTP applications
  effects of
    MAX_PDQPRIORITY 3-17
  effects of PDQ 12-12
  maximizing throughput with
    MAX_PDQPRIORITY 12-12, 12-15
  reducing
    DS_TOTAL_MEMORY 12-17
  using MGM to limit DSS resources 12-11
onaudit utility 5-60
ON-Bar, configuration parameters 5-57
oncheck utility
  and index sizing 7-9
  blobpage information 5-24
  checking index pages 7-21
  description of 2-16
  displaying
    data-page versions 6-57, 6-58
    free space 6-43
    free space in index 13-26
    page size 6-57
    size of table 6-12
  monitoring table growth 6-36
  obtaining information
    blobspaces 5-24, 5-27
  obtaining sbspace information 6-26
  outstanding in-place alters 6-57
  -pB option 5-24
  -pB option 2-16
  -pe option 6-43
  -pe option 2-16, 6-26, 6-41
  physical layout of chunk 6-41
  -pK option 2-16
  -pk option 2-16
  -pL option 2-17
  -pl option 2-16
  -pP option 2-17
  -pp option 2-17
  -pr option 6-57
  -pr option 2-17
  -pS option 6-27
  -pS option 6-27
  -ps option 2-17
  -pT option 6-57, 6-58
  -pT option 2-17
  -pt option 2-17, 6-12
ONDBSPDOWN configuration parameter 5-47
Online help Intro-14
Online manuals Intro-14
onload and onunload utilities 5-58, 6-6, 6-42, 6-45
onlog utility 1-8, 2-18
onmode utility
  and forced residency 4-23
  and PDQ 12-27
  and shared-memory connections 3-4
  -e option 13-43, 13-44
  –F option 4-11
  flushing SQL statement cache 13-43
  -MQDS options 12-14
  -P option 3-15
  -p option 3-27
  -W option
    changing STMT_CACHE_HITS 4-45
    changing STMT_CACHE_NOLIMIT 4-49
    changing STMT_CACHE_SIZE 4-47
ON-Monitor utility 3-27, 5-21
onparams utility 5-8, 5-11
onperf utility
  activity tools 14-21
  data flow 14-4
  description of 14-3
  graph tool 14-9
  metric classes
    database server 14-23
    disk chunk 14-25
    disk spindle 14-26
    fragment 14-30
    physical processor 14-26
    session 14-27
    tblspace 14-29
    virtual processor 14-27
  metrics 14-22
  monitoring tool 2-7
  query-tree tool 14-17
  replaying metrics 14-5
  requirements 14-6
  saving metrics 14-4
  starting 14-8
  status tool 14-20
  tools 14-6
  user interface 14-9
onspaces utility
  and blobspaces 5-21
  and sbspaces 5-28, 5-33, 6-30
  -ch option 6-30
  -Df BUFFERING tag 5-33
  -Df option 5-31, 6-33
  EXTENT_SIZE flag for sbspaces 5-31
  -S option 6-33
  smart large objects 6-30
  specifying lightweight I/O 5-33
  -t option 5-14, 5-19, 6-10, 7-19
onstat
  -g glo option 3-33
onstat utility
  -- option 2-9
  -a option 2-9
  -B option 4-57, 5-57
  -b option 2-10, 4-5, 4-15, 4-57, 5-45, 6-13
  -c option 6-17
  -d option 2-15, 3-30, 6-22
  -F option 5-55
  -g act option 2-11, 13-52
  -g afr option 3-26
  -g ath option 2-11, 3-11, 12-28, 13-52, 13-54
  -g cac option 10-43
  -g cac stmt option 4-44
  -g dic option 4-30, 4-31, 4-36, 6-39, 6-40
  -g dsc option 4-35, 4-37
  -g glo option 2-12
  -g iof option 2-15
  -g iog option 2-15
  -g ioq option 2-15, 3-15, 3-30
  -g iov option 2-15
  -g mem option 2-13, 4-7, 13-52, 13-55

-g mgm option 2-13, 12-11, 12-23
-g ntd option 2-12
-g ntf option 2-12
-g ntm option 3-25
-g ntu option 2-12, 3-25
-g option 2-10
-g ppf option 9-39
-g prc option 10-43, 10-44
-g qst option 2-12
-g rea option 2-11, 3-27, 3-28, 3-29
-g sch option 2-12
-g scn to monitor light scans 5-39
-g seg option 2-13, 4-24, 4-57
-g seg option 4-9
-g ses option 2-14, 2-20, 3-11, 4-8, 12-29, 13-46, 13-47, 13-48, 13-49, 13-52, 13-55
-g sle option 2-11
-g smb option 6-25
-g smb s option 6-29
-g spi option 2-12, 4-42, 4-50
-g sql option 2-19, 2-20, 13-49
-g sql session-id option 2-19, 13-58
-g ssc 4-42
-g ssc all 4-42
-g ssc option 4-43, 4-44, 4-45, 4-46, 4-51, 4-53, 13-50, 13-51
-g ssc output description 4-53
-g ssc pool option 4-51, 4-52
-g stm option 2-14, 2-20, 4-8, 4-54, 4-55, 13-46, 13-49, 13-52, 13-55
-g sts option 2-12, 4-8
-g tpf option 2-12
-g wai option 2-12
-g wst option 2-12
introduced 2-9
-k 2-19, 13-58
-k option 8-16, 8-17, 8-20, 8-27
-l option 2-10
-m option 5-43
monitoring AIO virtual processors 3-30
monitoring buffer use 4-17
monitoring byte locks 8-15
monitoring PDQ 12-23
monitoring sessions 13-52
monitoring SQL statement cache 4-42
monitoring tblspaces 6-38
monitoring transactions 2-19, 13-58, 13-59
monitoring virtual processors 3-28, 3-29, 3-33
-O option 5-35
-P option 2-10, 4-57
-p option 1-8, 2-10, 4-17, 4-57, 4-58, 8-18, 8-21
-R option 2-10, 5-56
-s option 4-59
-t option 6-38
-u 2-19, 13-52, 13-53, 13-58, 13-59
-u option 2-10, 2-19, 4-8, 5-44, 5-50, 8-19, 8-20, 12-27
-x option 2-10, 2-19
ontape utility 5-58
ON_RECVRY_THREADS configuration parameter 5-58
Opaque data type 7-22
OPCACHEMAX configuration parameter
description of 5-36
monitoring 5-35
Operating system
configuration parameters 3-3
file descriptors 3-6
NOFILE, NOFILES, NFILE, or NFILES configuration parameter 3-6
semaphores 3-4
SHMMAX configuration parameter 4-10
SHMMNI configuration parameter 4-11
SHMSEG configuration parameter 4-11
SHMSIZE configuration parameter 4-10
timing commands 1-12
Operator class, definition of 7-25, 7-31
OPTCOMPIND
and directives 11-18
configuration parameter 3-8, 3-16, 12-20
effects on query plan 10-26
environment variable 3-8, 3-16, 12-20
preferred join plan 12-20
Optical Subsystem 5-34
Optimization goal
default total query time 13-34
precedence of settings 13-35
setting with directives 11-11, 13-35
total query time 13-33 to 13-36
user-response and fragmented indexes 13-37
user-response time 13-33, 13-34, 13-35, 13-36
Optimization level
default 13-32
setting to low 13-32
table scan versus index scan 13-36
Optimizer
and hash join 10-6
and optimization goal 11-11, 13-34
and SET OPTIMIZATION statement 13-32, 13-35
autoindex path 13-23
choosing query plan 11-4, 11-6
composite index use 13-23
data distributions used by 13-15
index not used by 13-8
specifying high or low level of optimization 13-32
Optimizer directives
ALL_ROWS 11-11
altering query plan 11-12
and OPTCOMPIND 11-18
AVOID_EXECUTE 13-6
AVOID_FULL 11-5, 11-8
AVOID_HASH 11-11
AVOID_INDEX 11-8
AVOID_NL 11-5, 11-11
configuration parameter 11-18
effect on views 11-9, 11-10
EXPLAIN 11-16, 13-6
explain 11-16
EXPLAIN AVOID_EXECUTE 11-16
FIRST_ROWS 11-11, 11-12

FULL 11-8
guidelines 11-5
INDEX 11-8
influencing access plan 11-8
join order 11-7, 11-8
join plan 11-10
ORDERED 11-7, 11-8, 11-9, 11-10
purpose of 11-3
SPL routines 11-18
table scan 11-7
types 11-7
USE_HASH 11-10
USE_NL 11-10
OPT_GOAL
configuration parameter 13-34
environment variable 13-34
ORDER BY clause 10-25, 13-30
Ordered merge 13-37
Outer join, effect on PDQ 12-10
Outer table 10-5
Output description
onstat -g ssc 4-53
Outstanding in-place alters
definition of 6-57
displaying 6-58
performance impact 6-58

P

Page buffer
effect on performance 10-30
restrictions with simple large objects 6-19
Page size 4-5, 4-15, 5-45, 6-13
Pages
cleaning 5-40, 5-55
in memory 1-18
obtaining size 6-13
Paging
description of 1-18
DS_TOTAL_MEMORY 12-17
expected delay 1-19
memory-resident tables 4-56
monitoring 2-4, 4-17
RA_PAGES configuration parameter 5-40
RA_THRESHOLD configuration parameter 5-40
RESIDENT configuration parameter 4-4
Parallel
access to table and simple large objects 5-21, 5-28
backup and restore 5-57
execution of UDRs 13-38
index builds 12-7
inserts and DBSPACETEMP 5-14, 12-6
joins 12-15
scans 12-30, 13-39
sorts
and PDQ priority 13-30
when used 5-17
Parallel database queries (PDQ)
allocating resources for 12-12
and fragmentation 9-3
and remote tables 12-10
and SQL 9-3
and triggers 12-6, 12-8, 12-9
controlling resources for 12-21
effect of table fragmentation 12-3
gate information 12-26
how used 12-5
MGM resources 12-27
monitoring resources allocated for 12-23, 12-26, 12-27
priority
effect of remote database 12-10
queries that do not use PDQ 12-8
scans 3-18
SET PDQPRIORITY statement 12-20
SPL routines 12-9
statements affected by PDQ 12-9
user-defined routines 13-39
uses of 12-3
Parallel processing
MGM control of resources 12-10
ON-Bar 5-57
PDQ threads 12-3
user-defined routines 13-38
with fragmentation 9-15, 12-3
Parallel UDRs
description of 12-8, 13-38
enabling 13-39
sample query 13-39
when to use 13-38
Partitioning
definition of 9-3
See also Fragmentation.
PC_HASHSIZE configuration parameter 4-28, 10-42, 10-44
PC_POOLSIZE configuration parameter 4-28, 10-42, 10-44
PDQ priority
-1 value 12-13
DEFAULT tag 12-13
determining parallelism 12-16
effect of remote database 12-16
effect on parallel execution 12-13
effect on sorting memory 7-18
maximum parallelism 12-15
outer joins 12-16
parallel execution limits 12-16
SET PDQPRIORITY statement 12-20
SPL routines 12-16
See also PDQPRIORITY.
PDQPRIORITY
environment variable
requesting PDQ resources 12-12
limiting PDQ priority 12-14
See also PDQ priority.
PDQPRIORITY environment variable
adjusting value 12-14
for UPDATE STATISTICS 13-22
limiting PDQ priority 12-13, 12-14
limiting resources 3-8
overriding default 12-13
parallel sorts 13-30
setting PDQ priority 7-18
PDQPRIORITY parameter, effect of outer joins 12-10
PDQ. *See* Parallel database queries (PDQ).

Peak loads 1-14
Performance
  basic approach to measurement and tuning 1-4
  capturing data 2-6
  contiguous extents 5-30, 6-34
  dropping indexes for updates 7-17
  dropping indexes to speed modifications 6-50
  effect of
    contiguous disk space 5-30, 6-25, 6-34
    contiguous extents 6-41
    data mismatch 10-35
    disk access 10-31, 13-28
    disk I/O 5-5
    duplicate keys 7-14
    filter expression 13-8
    filter selectivity 10-23
    indexes 7-13 to 7-14
    redundant data 6-66
    regular expressions 13-8
    sequential access 13-28
    simple-large-object location 6-18
    table size 13-29
  goals 1-6
  improved by
    contiguous extents 5-30, 6-34
    specifying optimization level 13-32
    temporary table 13-31
  index time during modification 7-12
  measurements of 1-7
  slowed by data mismatch 10-35
  slowed by duplicate keys 7-15
Performance enhancements Intro-6
Performance problems
  early indications 1-4
  sudden performance loss 14-22
PHYSBUFF configuration parameter 4-21, 5-29, 5-48
PHYSFILE configuration parameter 5-44
Physical log
  buffer size 4-21
  configuration parameters that affect 5-12
  effects of checkpoints on sizing 5-46
  effects of frequent updating 5-46
  increasing size 5-44, 5-47
  mirroring 5-11
Platform icons Intro-12
Playback process 14-5
Poll thread
  added with network VP 3-27
  configuring with NETTYPE 3-4, 3-19
  connections per 3-20
  NETTYPE configuration parameter 3-21
Priority, setting on Windows 2000 3-7
Priority, setting on Windows NT 3-7
Probe table, directives 11-10, 11-11
Product icons Intro-12
Program group
  Documentation notes Intro-16
  Release notes Intro-16
PSORT_DBTEMP environment variable 5-16
PSORT_NPROCS
  environment variable 5-17
PSORT_NPROCS environment variable 3-9, 7-18, 13-30

Q

Quantum, of memory 3-17, 4-18, 12-11, 12-13, 12-22, 12-23, 12-25
Queries (DSS)
  and temporary files 9-10, 13-31
  resources allocated to 12-22
  response time and throughput 1-11
Query optimizer. *See* Optimizer.
Query plan 10-3
  all rows 11-11
  altering with directives 11-7, 11-12, 11-14, 11-15
  autoindex path 13-23
  avoid query execution 11-16
  chosen by optimizer 11-4, 11-6
  collection-derived table 10-20
  disk accesses 10-10
  displaying 10-12
  displaying all 13-27
  first-row 11-11
  fragment elimination 9-40, 12-30
  in pseudocode 10-8 to 10-10
  join order 11-14
  restrictive filters 11-4
  row access cost 10-30
  time costs of 10-7, 10-27, 10-28, 10-30
  use of indexes in 10-10
Query-tree tool, onperf 14-6

R

Range expression, definition of 9-23
Raw devices. *See* Unbuffered devices.
Raw files. *See* Unbuffered files.
RA_PAGES configuration parameter 5-40, 5-55
RA_THRESHOLD configuration parameter 5-40, 5-55
Read cache rate 4-17
Read-ahead
  configuring 5-38, 5-40
  description 5-37
Recent history 14-17
Reclaiming empty extent space 6-44
Redundant data, introduced for performance 6-66
Redundant pairs, definition of 10-12
Regular expression, effect on performance 13-8

Relational model,
denormalizing 6-61
Release notes Intro-15
Release notes, program
item Intro-16
Remainder pages, for tables 6-14
Remote database, effect on
PDQPRIORITY 12-10
RENAME statement 10-40
Repeatable Read isolation
level 5-39, 8-11, 10-26
Residency 4-23
RESIDENT configuration
parameter 4-23
Resident portion of shared
memory 4-4
Resizing table to reclaim empty
space 6-44
Resource utilization
and performance 1-14
capturing data 2-5
CPU 1-17
description of 1-15
disk 1-20
factors that affect 1-22
memory 1-18
operating-system resources 1-15
Resource, critical 1-15
Response time
actions that determine 1-10
contrasted with throughput 1-11
improving with
MaxConnect 3-34
improving with multiplexed
connections 3-33
measuring 1-12
Response times
SQL statement cache 13-41
RETURN statement 10-42
Root dbspace, mirroring 5-9
Root index page 7-5
ROOTNAME configuration
parameter 5-11
ROOTOFFSET configuration
parameter 5-11
ROOTPATH configuration
parameter 5-11
ROOTSIZE configuration
parameter 5-11
Round-robin distribution
scheme 9-11 to 9-12
Round-robin fragmentation, smart
large objects 9-11
Row access cost 10-30
Row pointer
attached index 9-18
detached index 9-19
in fragmented table 9-10
space estimates 7-6, 9-10
R-tree index
description of 7-10, 7-27
physical log size 5-44
usage 7-23, 7-24

S

Sample-code conventions Intro-13
sar command 2-6, 4-17
Saturated disks 5-6
Sbspace
changing storage
characteristics 6-30
configuration impacts 5-27
creating 5-31, 6-30
description of 5-9
estimating space 6-20
extent 5-30, 5-31, 5-33
inherited storage
characteristics 6-31
metadata requirements for 6-20
metadata size 6-22
monitoring 6-25
monitoring extents 6-26, 6-27
SBSPACENAME configuration
parameter 6-31
Sbspace extents, performance 5-30,
6-24
SBSPACENAME
configuration parameter 5-19
logging 5-19
SBSPACENAME configuration
parameter 5-28, 6-31, 6-32
SBSPACETEMP
configuration parameter 5-18,
5-19, 5-20
no logging 5-18, 5-19
Scan threads 3-18
Scans
DS_MAX_QUERIES 12-11
DS_MAX_SCANS 12-11
first-row 10-19
key-only 10-4
light 5-38, 5-39
lightweight I/O 5-32
limited by
MAX_PDQPRIORITY 3-16
limiting number of 12-18
memory-management
system 1-18
parallel 12-29
parallel database query 3-18
RA_PAGES and
RA_THRESHOLD 5-40
read-ahead I/O 5-38
sequential 5-37
skip-duplicate-index 10-19
table 10-4, 10-5, 13-23
threads 3-11, 3-16, 3-18
Scheduling facility, cron 2-7, 4-12
Screen-illustration
conventions Intro-14
Secondary access method
DataBlade module 7-23
defined by database server 7-24
description of 7-23 to 7-27
generic B-tree 7-24
R-tree 7-27
Seek time, disk I/O 10-30
SELECT statement
accessing data 9-8
collection-derived tables 10-19
column filter 10-9
COUNT clause 10-9
join order 10-7
materialized view 10-46
redundant join pair 10-12
row size 6-13
SPL routines and directives 11-18
three-way join 10-8
trigger performance 10-46
triggers 10-46
using directives 11-3 to 11-5
Selective filter
on dimensional table 13-25

Selectivity
and indexed columns 7-15
column, and filters 7-14
definition of 10-23
estimates for filters 10-23
user-defined data 13-38, 13-39, 13-40
Semaphores, allocated for UNIX 3-4
Semi join, description of 10-6
SEMMNI UNIX configuration parameter 3-4, 3-5
SEMMNS UNIX configuration parameter 3-5
SEMMSL UNIX configuration parameter 3-4
Sequential
access costs 10-31
scans 5-37, 13-28
Service time formula 1-16
Session
monitoring 2-20, 13-52
monitoring memory 4-8, 4-54, 4-55
setting optimization goal 13-35
Sessions
monitoring 13-52, 13-55
monitoring memory 13-52, 13-55
SET DATASKIP statement 9-7
SET EXPLAIN
and PDQ priority levels 12-30
directives 11-13, 11-15
output
sqexplain.out (UNIX) 10-13
sqexpln file (Windows NT) 10-13
output explanation 10-13
to determine UPDATE STATISTICS 13-18
to show
collection scan 10-20
complex query 10-15
converted data 10-35
data mismatch 10-35
decisions of query optimizer 12-19
directives 11-13, 11-15
estimated cost of query 10-13
estimated number of rows 10-13
fragments scanned 9-40
how data accessed 9-8
join rows returned 13-18
key-first scan 10-17
optimization 13-36
optimizer access paths 10-17
order of tables accessed 10-17
parallel scans 12-29
query plan 10-12, 12-19
remote access plan 10-14
resources required by query 13-6
secondary threads 12-29
serial scans 12-29
simple query 10-15
SPL routine 10-39
subquery 10-18
temporary table for views 10-34
using 10-12 to 10-18
SET INDEX statement
buffer pages 4-57
memory resident 4-56
SET ISOLATION statement 8-9, 8-14
SET LOCK MODE statement 8-4, 8-9, 8-12, 8-16, 8-19, 8-21
SET LOG statement 1-8
SET OPTIMIZATION statement
setting ALL_ROWS 13-35
setting FIRST_ROWS 13-35
setting HIGH or LOW 13-32
SPL routines 10-41
SET PDQPRIORITY statement
DEFAULT tag 12-13, 12-20
in application 12-13, 12-20
in SPL routine 12-16
limiting CPU VP utilization 3-9
sort memory 13-22
SET STATEMENT CACHE statement 4-40, 13-44, 13-45
SET TABLE statement
buffer pool pages 4-57
memory-resident tables 4-56
SET TRANSACTION statement 8-9
Shared memory
allowed per query 4-18
amount for sorting 7-19
communication interface 3-4
database server portion 4-3
freeing 4-11
message portion 4-9
resident portion 4-4
size limit 4-24
size of added increments 4-24
virtual portion 4-6, 4-13
SHMADD configuration parameter 4-6, 4-24
SHMBASE configuration parameter 4-12
SHMMAX operating-system configuration parameter 4-10
SHMMNI operating-system configuration parameter 4-11
SHMSEG operating-system configuration parameter 4-11
SHMSIZE operating-system configuration parameter 4-10
SHMTOTAL configuration parameter 4-4, 4-6, 4-24
SHMVIRTSIZE configuration parameter 4-6, 4-25
Short rows, reducing disk I/O 6-62
Simple large objects
blobpage size 5-23
blobspace 5-20
configuration effects 5-20
disk I/O 5-22
estimating number of blobpages 6-17
estimating tblspace pages 6-19
how stored 6-18
in blobspace 5-21
in dbspace 6-12
locating 6-18
logging 5-21
logical-log size 5-51
Optical Subsystem 5-35
parallel access 5-21
SINGLE_CPU_VP configuration parameter 3-11
Smart large object
buffering recommendation 5-33
extent size 5-30, 5-32

Smart large objects
  ALTER TABLE 6-33
  buffer pool 4-16, 5-29, 5-32
  buffer pool usage 6-32
  changing characteristics 6-33
  CREATE TABLE statement 6-33
  DataBlade API functions 5-27, 5-31, 6-31, 6-37, 8-28
  disk I/O 5-27
  ESQL/C functions 5-27, 5-31, 6-31, 6-37, 8-28
  estimated size 6-32
  estimating space 6-20
  extent size 6-31, 6-37
  fragmentation 6-32, 9-11
  I/O operations 5-33, 6-24
  I/O performance 4-16, 5-29, 5-32, 5-34, 6-23, 6-24
  last-access time 6-31
  lightweight I/O 4-16, 5-32 to 5-34
  lock mode 6-31
  logging status 6-31
  logical-log size 5-51
  minimum extent size 6-32
  mirroring chunks 5-9
  monitoring 6-25
  next-extent size 6-31
  sbspace 5-27
  sbspace name 6-32
  setting isolation levels 8-28
  specifying characteristics 6-33
  specifying size 5-31, 6-32, 6-37
  storage characteristics 6-30
SMI table
  monitoring sessions 13-57
  monitoring virtual processors 3-31
SMI tables
  monitoring latches 4-59
Snowflake schema 13-24
Software dependencies Intro-4
Sorting
  avoiding with temporary table 13-31
  costs of 10-28
  DBSPACETEMP configuration parameter 5-12
  DBSPACETEMP environment variable 5-12
  effect of PDQ priority 13-22
  effect on performance 13-29
  effect on SHMVIRTSIZE 4-7
  estimating temporary space 7-20
  memory estimate 7-19
  PDQ priority for 7-20
  query-plan cost 10-4
  sort files 2-15, 5-12
  space pool 4-6
  triggers in a table hierarchy 10-46
SPL routines
  automatic reoptimization 10-40
  display query plan 10-39
  effect of PDQ 12-9
  effect of PDQ priority 12-16
  optimization level 10-41
  query response time 1-10
  when executed 10-41
  when optimized 10-38
sqexplain.out file 10-13
sqexpln file 10-13
SQL
  new features Intro-8
SQL code Intro-13
SQL statement cache
  cleaning 4-41, 4-46
  description 13-41
  effect on prepared statements 13-42
  effect on SHMVIRTSIZE 4-7
  exact match 13-46
  flushing 13-43
  hits 4-28, 4-40, 4-41, 4-42, 4-43, 4-44, 4-45, 4-48, 4-49, 4-50, 4-53, 4-54
  host variables 13-42
  memory 4-28
  memory limit 4-41, 4-48
  memory size 4-41
  monitoring 4-42, 4-43, 4-44, 4-45, 4-53
  monitoring dropped entries 13-50, 13-51
  monitoring pools 4-50, 4-51, 4-52
  monitoring session memory 13-46, 13-47, 13-48, 13-49
  monitoring size 4-46, 4-51
  monitoring statement memory 2-20, 13-46, 13-49
  number of pools 4-50, 4-51
  performance benefits 4-37, 13-41
  response times 13-41
  size 4-28, 4-46, 4-48
  specifying 13-44, 13-45
  STMT_CACHE configuration parameter 4-39, 13-44, 13-45
  STMT_CACHE environment variable 13-44, 13-45
  STMT_CACHE_HITS configuration parameter 4-49
  STMT_CACHE_SIZE configuration parameter 4-47, 4-49
  when to use 13-42
SQLCODE field of SQL Communications Area 6-63
sqlhosts file
  client buffer size 3-26
  multiplexed option 3-33
sqlhosts file or registry
  connection type 3-19, 3-20
  connections 4-25
  number of connections 5-50
  one connection type 3-22
SQLWARN array 5-41
Stack
  specifying size 4-26
STACKSIZE configuration parameter 4-26
STAGEBLOB configuration parameter
  description 5-36
  monitoring 5-35
Staging area, optimal size for blobspace 5-36
Star join, description of 13-25
Star schema 13-24
Statement cache. *See* SQL Statement cache.

Status tool 14-6, 14-20
STMT_CACHE configuration parameter 13-44, 13-45
STMT_CACHE environment variable 13-44, 13-45
STMT_CACHE_HITS configuration parameter 4-28, 4-40, 4-41, 4-42, 4-43, 4-44, 4-45, 4-48, 4-49, 4-50, 4-53, 4-54
STMT_CACHE_NOLIMIT configuration parameter 4-28, 4-41, 4-48
STMT_CACHE_NUMPOOL configuration parameter 4-50, 4-51
STMT_CACHE_SIZE configuration parameter 4-28, 4-41, 4-46, 4-48
Storage characteristics, system default 6-31
Storage statistics
  blobpages 5-24
  blobspaces 5-24
Stored Procedure Language (SPL) 10-41
Stored procedure. *See* SPL routine.
stores_demo database Intro-5
Strategy function, for secondary-access methods 7-32
Strings, expelling long 6-62
Structured Query Language (SQL)
  ALTER FRAGMENT statement 6-6, 6-45
  ALTER FUNCTION statement, parallel UDRs 13-39
  ALTER INDEX statement 6-43, 6-45, 7-15
    compared to CREATE INDEX 6-44
    TO CLUSTER clause 6-43
  ALTER TABLE statement 6-35, 6-44
    changing extent sizes 6-37
    changing sbspace characteristics 6-33
    PUT clause 6-33
    sbspace fragmentation 9-11
  COMMIT WORK statement 1-8
  CONNECT statement 5-7, 6-6
  CREATE CLUSTERED INDEX statement 7-16
  CREATE FUNCTION statement 3-10
    parallel UDRs 13-39
    selectivity and cost 13-40
    specifying stack size 4-26
  CREATE INDEX statement
    attached index 9-17
    compared to ALTER INDEX 6-44
    detached index 9-18
    generic B-tree index 7-24
    TO CLUSTER clause 6-43
  CREATE PROCEDURE statement, SQL optimization 10-38
  CREATE TABLE statement
    blobspace assignment 5-21
    extent sizes 6-35
    fragmentation 9-17, 9-18
    IN DBSPACE clause 6-6
    lock mode 8-7, 8-8
    PUT clause 6-33
    sbspace characteristics 6-33
    sbspace fragmentation 9-11
    simple large objects 6-18
    system catalog table 5-7
    TEMP TABLE clause 5-12, 5-19
  CREATE TEMP TABLE statement 9-20
  DATABASE statement 5-7, 6-6
  DISTINCT keyword 13-24
  EXECUTE PROCEDURE statement 10-41
  EXTENT SIZE clause 6-35
  FRAGMENT BY EXPRESSION clause 9-17
  GROUP BY clause 10-25
    MGM memory 12-11
  IN DBSPACE clause 6-6
  INSERT statement 9-11
  LOAD and UNLOAD statements 6-6, 6-42, 6-45, 7-17
  MODIFY NEXT SIZE clause 6-35, 6-37
  NEXT SIZE clause 6-35
  optimizer directives 11-7
  ORDER BY clause 10-25
  RENAME statement 10-40
  SELECT statement
    collection-derived tables 10-19
    column filter 10-9
    COUNT clause 10-9
    join order 10-7
    materialized view 10-46
    redundant join pair 10-12
    row size 6-13
    SPL routines and directives 11-18
    three-way join 10-8
    triggers 10-46
    using directives 11-3 to 11-5
  SET DATASKIP statement 9-7
  SET EXPLAIN directives 11-13
  SET EXPLAIN statement 9-40
    accessing data 9-8
    collection scan 10-20
    complex query 10-15
    estimated number of rows 10-13
    flattened subquery 10-18
    how data accessed 9-8
    materialized view 10-34
    optimizer decisions 12-19
    order of tables 10-17
    output explanation 10-13
    show query plan 10-12
    simple query 10-15
  SET EXPLAIN statement, directives 11-15
  SET INDEX statement
    buffer pages 4-57
    memory resident 4-56
  SET ISOLATION statement 8-9
  SET LOCK MODE statement 8-4, 8-9, 8-12, 8-16, 8-19, 8-21
  SET OPTIMIZATION statement 13-32, 13-35
  SET PDQPRIORITY statement 3-9
    DEFAULT tag 12-13, 12-20
    in application 12-13, 12-20
    in SPL routine 12-16
    sort memory 13-22
  SET STATEMENT CACHE 4-40, 13-44, 13-45

SET TABLE statement
buffer pool pages 4-57
memory-resident tables 4-56
SET TRANSACTION
statement 8-9
TO CLUSTER clause 6-43, 6-45
UPDATE STATISTICS
statement 4-6, 10-21, 11-6
and directives 11-6, 11-18
creating data
distributions 13-15
data distributions 10-22
effect of PDQ 12-9
guidelines to run 13-13 to 13-22
HIGH mode 13-13, 13-16, 13-18, 13-19, 13-20
LOW mode 13-13, 13-14, 13-17, 13-38
MEDIUM mode 13-16, 13-19
multiple column
distributions 13-22
on join columns 13-18
on user-defined data
columns 13-19
optimizing SPL routines 12-16
query optimization 13-13
reoptimizing SPL
routines 10-40
updating system catalog 10-21, 13-13
user-defined data 13-38
WHERE clause 10-25, 13-7, 13-8
Subqueries 12-16
flattening 10-18
rewriting 10-18
superstores Intro-5
superstores_demo database Intro-5
Support function, description for
secondary access method 7-32
Swap device 1-19
Swap space 1-18, 4-11
Swapping, memory-management
scheme 1-19, 12-17
Symbol table, building 6-63
sysprofile table 8-18
System catalog tables
data distributions 10-22
optimizer use of 10-22
sysams 7-25, 7-35
syscolumns 13-15, 13-19
sysdistrib 13-15, 13-19
sysfragments 9-19, 9-40
sysprocbody 10-38
sysprocedure 10-38
sysprocplan 10-38, 10-40
systables 7-21, 10-40
systrigbody 10-45
systriggers 10-45
updated by UPDATE
STATISTICS 10-21
System requirements
database Intro-4
software Intro-4
System resources, measuring
utilization of 1-15
System-monitoring interface
(SMI) 2-6, 2-7

T

Table hierarchy, SELECT
triggers 10-46
Table scan
and OPTCOMPIND 3-16
description of 10-4
nested-loop join 10-5
replaced with composite
index 13-23
Tables
adding redundant data 6-66
and middle partitions of disks 6-8
assigning to dbspace 6-6
companion, for long strings 6-63
configuring I/O for 5-37
cost of access 13-29
cost of companion 6-65
denormalizing 6-62
dimensional 13-24
division by bulk 6-64
estimating
blobpages in tblspace 6-17
data page size 6-12
size with fixed-length rows 6-12
size with variable-length
rows 6-15
expelling long strings 6-62
fact 13-24, 13-25
frequently updated
attributes 6-65
infrequently accessed
attributes 6-64
isolating high-use 6-7
locks 8-6, 8-7
managing
extents 6-34
indexes for 7-11
memory-resident 4-57
nonfragmented 6-11
partitioning, definition of 9-3
placement on disk 6-5
reducing contention between 6-7
redundant and derived data 6-65
remote, used with PDQ 12-10
rows too wide 6-64
shorter rows 6-62
size estimates for 6-12
storage on middle partitions of
disk 6-8
temporary 6-10
TAPEBLK configuration
parameter 5-58
TAPEDEV configuration
parameter 5-58
TAPESIZE configuration
parameter 5-58
Tblspace
attached index 9-19
definition of 6-12
monitoring active tblspaces 6-38
simple large objects 6-18
TCP/IP buffers 3-22
TEMP or TMP user environment
variable 5-13
TEMP TABLE clause of the
CREATE TABLE
statement 5-12, 5-19, 9-20
Temporary dbspaces
creating 7-19
DBSPACETEMP configuration
parameter 5-15
DBSPACETEMP environment
variable 5-15
decision-support queries 9-9
for index builds 7-19, 7-20

in root dbspace 5-13
monitoring 2-15
onspaces -t 5-14
optimizing 5-14
Temporary sbspaces
configuring 5-18
onspaces -t 5-19
optimizing 5-19
SBSPACETEMP configuration parameter 5-20
Temporary smart large objects
LO_TEMP flag 5-18
Temporary tables
configuring 5-12
DBSPACETEMP configuration parameter 5-12, 5-15
DBSPACETEMP environment variable 5-15
decision-support queries 9-9
explicit 9-20
fragmentation 9-20
in root dbspace 5-7
speeding up a query 13-31
views 10-34
TEXT data type 6-63
buffer pool restriction 6-19
how stored 6-18
in blobspace 5-20
in table-size estimate 6-13
locating 6-18
memory cache 5-34
on disk 6-18
parallel access 5-28
staging area 5-35
See also Simple large objects.
Thrashing, definition of 1-18
Thread
control blocks 4-6
DS_MAX_SCANS configuration parameter 12-11
MAX_PDQPRIORITY 3-16
monitoring 2-11, 2-19, 4-8, 13-52, 13-54
page-cleaner 5-11
primary 12-3, 12-28
secondary 12-3, 12-29
sqlexec 2-11, 5-55, 12-28
Thread-safe UDRs 13-39
Throughput
benchmarks 1-8
capturing data 1-8
contrasted with response time 1-11
discussion of 1-7
measured by logged COMMIT WORK statements 1-8
Tightly coupled 13-60, 13-62
Tightly coupled mode 13-60
Time
getting current in ESQL/C 1-13
getting user, processor and elapsed 1-13
getting user, system, and elapsed 1-12
time command 1-12
Timing
commands 1-12
functions 1-13
performance monitor 1-13
Tip icons Intro-12
TO CLUSTER clause 6-43, 6-45
TPC-A, TPC-B, TPC-C, and TPC-D benchmarks 1-9
Transaction
cost 1-14
loosely-coupled mode 13-59, 13-60
monitoring 2-19, 13-58
monitoring global transactions 13-60, 13-62
rate 1-7
rollback 7-13
tightly-coupled mode 13-59, 13-60, 13-62
Transaction Processing Performance Council (TPC) 1-8
Transaction throughput, effects of MAX_PDQPRIORITY 3-16
Transactions
monitoring 2-19, 13-52, 13-53, 13-58, 13-59
Triggers
and PDQ 12-6, 12-8, 12-9
behavior in table hierarchy 10-46
description of 10-44
effect of PDQ 12-9
performance 10-45
row buffering 10-46

U

UDR cache
buckets 4-28, 10-42, 10-44
number of entries 4-28, 10-42, 10-44
Unbuffered devices 10-31
Unbuffered logging 5-10
UNIX
cron scheduling facility 2-7
default locale for Intro-5
iostat command 2-6
network protocols 3-19
ps command 2-6
sar command 2-6
SEMMNI configuration parameter 3-4, 3-5
SEMMNS configuration parameter 3-5
SEMMSL configuration parameter 3-4
sqexplain.out file 10-13
time command 1-12
vmstat command 2-6
Update cursor 8-13
UPDATE STATISTICS statement
and directives 11-6, 11-18
creating data distributions 13-15
data distributions 10-22
effect of PDQ 12-9
effect on virtual portion of memory 4-6
generating with ISA 13-13
guidelines to run 13-13 to 13-22
HIGH mode 11-6, 13-13, 13-16, 13-18, 13-19, 13-20
improving ALTER FRAGMENT ATTACH performance 9-34, 9-35

LOW mode 13-13, 13-14, 13-17, 13-38
MEDIUM mode 13-16, 13-19
multiple column distributions 13-22
on join columns 13-18
on user-defined data columns 13-19
optimizing SPL routines 12-16
providing information for query optimization 10-21
query optimization 13-13
reoptimizing SPL routines 10-40
updating system catalog 10-21, 13-13
user-defined data 13-38, 13-40
Usability enhancements Intro-7
User-defined aggregate, parallel execution 13-38
User-defined data type
B-tree index 7-10
cost of routine 13-38, 13-39, 13-40
data distributions 13-19
generic B-tree index 7-25
opaque 7-22
optimizing queries on 13-37
selectivity of 13-38, 13-40
UPDATE STATISTICS 13-19
User-defined index
DataBlade module 7-22
User-defined index, DataBlade module 7-11
User-defined routine cache
changing size 10-42
contents 10-42
effect on SHMVIRTSIZE 4-7
User-defined routines
enabling parallel execution 13-39
negator function 13-41
parallel execution 12-8, 13-38
query filters 13-7
query response time 1-10
stack size 4-26
statistics 13-40
thread-safe 13-39
User-defined selectivity function 13-7
User-defined statistics 13-40
Users, types of Intro-4
USING clause, CREATE INDEX statement 7-27
Utilities
database server performance measurement 2-7
DB-Access 6-42
dbload 6-42, 7-17
dbschema 9-9, 9-14, 13-20, 13-21, 13-22
ISA 4-59
and blobspaces 5-21
capabilities 2-8
creating staging-area blobspace 5-35
description 2-8
generating UPDATE STATISTICS statements 13-13
monitoring I/O Utilization 2-15
monitoring user sessions 2-19
starting virtual processors 3-27
onaudit 5-60
oncheck
and index sizing 7-9
introduced 2-16
monitoring table growth 6-36
-pB option 2-16
-pe option 2-16, 6-26, 6-41, 6-43
-pK option 2-16
-pk option 2-16
-pL option 2-17
-pl option 2-16
-pP option 2-17
-pp option 2-17
-pr option 2-17, 6-57
-pS option 6-27
-ps option 2-17
-pT option 2-17, 6-57, 6-58
-pt option 2-17, 6-12
onload and onunload 5-58, 6-6, 6-42, 6-45
onlog 1-8, 2-18
onmode
and forced residency 4-23
and shared-memory connections 3-4
-F option 4-11
-MQDS options 12-14
-P option 3-15
-p option 3-27
-W option to change STMT_CACHE_HITS 4-45
-W option to change STMT_CACHE_NOLIMIT 4-49
-W option to change STMT_CACHE_SIZE 4-47
onmode and PDQ 12-27
ON-Monitor 3-27, 5-21, 5-35
onparams 5-8, 5-11
onperf
activity tools 14-21
data flow 14-4
description of 14-3
graph tool 14-9
metrics 14-22
query-tree tool 14-17
replaying metrics 14-5
requirements 14-6
saving metrics 14-4
starting 14-8
status tool 14-20
tools 14-6
user interface 14-9
onspaces
and blobspaces 5-21
and sbspaces 5-28, 6-30
-ch option 6-30
creating staging-area blobspace 5-35
-Df BUFFERING tag 5-33
-Df option 5-31, 6-33
EXTENT_SIZE flag for sbspaces 5-31
-S option 6-33
-t option 5-14, 5-19, 6-10, 7-19
onstat
-- option 2-9
-a option 2-9
-B option 4-57, 5-57
-b option 2-10, 4-5, 4-15, 4-57, 5-45, 6-13
-c option 6-17
-d option 2-15, 3-30, 6-22
-F option 5-55
-g act option 2-11, 13-52
-g afr option 3-26

-g ath option 2-11, 3-11, 12-28, 13-52, 13-54
-g cac option 4-44, 10-43
-g cac stmt option 4-44
-g dic option 4-30, 4-31
-g dsc option 4-35, 4-37
-g glo option 2-12
-g iof option 2-15
-g iog option 2-15
-g ioq option 2-15, 3-15, 3-30
-g iov option 2-15
-g lsc option 5-39
-g mem option 2-13, 4-7, 13-52, 13-55
-g mgm option 2-13, 12-11, 12-23
-g ntd option 2-12
-g ntf option 2-12
-g ntm option 3-25
-g ntu option 2-12, 3-25
-g option 2-10
-g ppf option 9-39
-g prc option 10-43, 10-44
-g qst option 2-12
-g rea option 2-11, 3-27, 3-28, 3-29
-g sch option 2-12
-g seg option 2-13, 4-9, 4-24, 4-57
-g ses option 2-14, 2-20, 3-11, 4-8, 12-29, 13-52, 13-55
-g sle option 2-11
-g smb option 6-25
-g smb s option 6-29
-g spi option 2-12, 4-42
-g sql option 2-19, 2-20
-g ssc all option 4-42
-g ssc option 4-42, 13-51
-g stm option 2-14, 4-8, 4-54, 4-55, 13-52, 13-55
-g sts option 2-12, 4-8
-g tpf option 2-12
-g wai option 2-12
-g wst option 2-12
introduced 2-9
-k option 8-17, 8-20
-l option 2-10
-m option 5-43
monitoring buffer pool 4-17
monitoring threads per session 3-11
-O option 5-35
-P option 2-10, 4-57
-p option 1-8, 2-10, 4-17, 4-57, 4-58, 8-18, 8-21
-R option 2-10, 5-56
-s option 4-59
-u option 2-10, 2-19, 4-8, 5-44, 5-50, 8-19, 8-20, 12-27, 13-52, 13-53
-x option 2-10, 2-19
ontape 5-58
Utilization
capturing data 2-5
CPU 1-17, 3-3 to 3-34
definition of 1-15
disk 1-20
factors that affect 1-22
memory 1-18, 4-3 to 4-57
service time 1-15

V

VARCHAR data type
access plan 10-4
byte locks 8-15
cost of 10-36
expelling long strings 6-62
in table-size estimates 6-15
when to use 6-62
Variable-length rows 6-15
Views, effect of directives 11-9, 11-10
Virtual memory, size of 4-11
Virtual portion 4-6, 4-25
Virtual portion of shared memory
contents of 4-13
Virtual processors
and NETTYPE 3-20
Virtual processors (VPs)
adding 3-27
and CPU 3-27
class name 3-10
monitoring 3-27, 3-28, 3-29, 3-30
network, SOC or TLI 3-27
semaphores required for 3-4
setting number of CPU VPs 3-10
setting number of NET VPs 3-20
starting additional 3-27
user-defined 3-9
vmstat command 2-6, 4-17
VPCLASS configuration parameter
process priority aging 3-12
setting number of AIO VPs 3-15
setting number of CPU VPs 3-10
setting processor affinity 3-12, 3-13
specifying class of virtual processors 3-9

W

Warning icons Intro-12
WHERE clause 10-25, 13-7, 13-8
Windows NT
default locale for Intro-5
NETTYPE parameter 3-21
network protocols 3-19
parameters that affect CPU utilization 3-7
Performance Monitor 1-13, 2-6
sqexpln file 10-13
TEMP or TMP user environment variable 5-13
Write once read many (WORM) optical subsystem 5-34

X

X display server 14-7
X/Open compliance level Intro-17